OT-Help 2.0 User Guide[*†]

Kevin Mullin
Brian W. Smith
Joe Pater
John J. McCarthy
University of Massachusetts, Amherst

2010 June 22 Wednesday

Website: http://web.linguist.umass.edu/~OTHelp/

OT-Help Developer Contact

Bug Reports:	OTHelp.contact+bugs@gmail.com
Feature Requests:	OTHelp.contact+features@gmail.com
Questions:	OTHelp.contact+questions@gmail.com
Comments:	OTHelp.contact+comments@gmail.com

[*] This PDF document contains hyperlinks.

[†] This document was prepared with support from grant BCS–0813829 from the National Science Foundation to the University of Massachusetts, Amherst.

Contents

OT-Help 2.0 **3**

1 Serial Typology Calculation **4**
1.1 The Basics . . . 4
1.2 Treatment of Ties . . . 7
1.2.1 *Input Death* . . . 9

2 Using OT-Help 2.0 **11**
2.1 Platforms and System Requirements . . . 12
2.2 Program Files . . . 12
2.3 Installation . . . 12
2.4 Starting the Program . . . 13
2.5 User Files . . . 13
2.5.1 Input File Internal Format . . . 15
2.5.2 Operation File . . . 17
2.5.3 Constraint File . . . 19
2.5.4 Commenting . . . 23
2.6 Loading User Files . . . 23
2.7 Home Page Window . . . 25
2.8 Language Typology Window . . . 26
2.9 Derivation Windows . . . 29
2.10 Displaying Derivational Ties . . . 34
2.10.1 Multiple Outputs . . . 34
2.10.2 Input Death . . . 36
2.11 Printing . . . 39
2.12 Saving and Exporting . . . 41

3 Operations **41**
3.1 Operation Filename & Extension . . . 42
3.2 Operation File Format . . . 42
3.3 Operation Parameters . . . 43
3.4 Context-free Operations . . . 44
3.4.1 Character Replacement . . . 46
3.4.2 Character Deletion . . . 46
3.4.3 Character Insertion . . . 48
3.5 Context Dependency & Substring Indexing . . . 49

4 Constraints **52**
4.1 Constraint Filename & Extension . . . 53
4.2 Constraint File Format . . . 54
4.3 Format of User-defined Constraints . . . 54
4.3.1 Parameters . . . 55
4.4 Constraint Types . . . 59
4.4.1 `Markedness` . . . 59
4.4.2 `Alignment` . . . 62
4.4.3 `Scalar` . . . 68
4.4.4 `Long-distance` . . . 70

4.4.5 `Faithfulness` 73
4.4.6 Positive Constraints 74
4.4.7 Pre-defined Constraints 76

References **78**

OT-Help 2.0

OT-Help 2.0 (Staubs *et al.* 2009) is a Java-based open source software tool that aids research in four variants of Optimality Theory (OT; Prince & Smolensky 1993/2004). From OT-Help 1.2, it inherits the ability to find both constraint rankings, as in classic OT, and constraint weightings, as in Harmonic Grammar (HG; Legendre, Miyata & Smolensky 1990a, 1990b; see Smolensky & Legendre 2006 and Pater 2009 for overviews of subsequent research). OT-Help 2.0 adds the ability to calculate the typological predictions of a derivational variant of OT termed *Harmonic Serialism* by Prince & Smolensky (1993/2004); see McCarthy (submitted) for an introduction and survey of recent work. Serial typology calculations can also make use of weighted constraints, thus permitting exploration of a fourth variant of OT: serial HG (Pater to appear).

The tools for parallel OT and HG in OT-Help 2.0 remain the same as in OT-Help 1.2. Readers are directed to Becker and Pater (2007) for a user guide for that portion of the software, and to Potts *et al.* (2010) for a presentation of its linear programming method for finding HG weights, and for applications in comparing parallel OT and HG.[1]

The present guide focuses on the new capabilities of OT-Help 2.0: serial typology calculation and candidate generation and evaluation. Section 1 introduces the typology calculation method using a simple example, and section 2 shows how to run that example calculation in OT-Help. Section 3 provides further details on defining the operations that generate candidates, and section 4 provides further details on constraint definitions. We assume familiarity with serial and weighted constraint versions of OT; see McCarthy (submitted) and Pater (2009) for introductions.

[1] We plan to eventually incorporate the user guides for versions 1.2 and 2.0 into a single document.

1 Serial Typology Calculation

1.1 The Basics

This section describes how OT-Help 2.0 calculates typologies in serial OT and HG. We start with serial OT and a single input; we then generalize to serial HG and to multiple inputs.

From an input *i*, a candidate set is created that includes *i* and all of the candidates that can be generated with a single application of any of the operations defined in the operation file (see sections 2.5.2 and 3). The constraints in the constraint file (see sections 2.5.3 and 4) are applied to each candidate to create a violation vector consisting of the number of violations that candidate incurs from each constraint. The result is equivalent to an unranked traditional violation tableau such as that in (1). This tableau assumes an initial input /bakad/, operations that change voiced stops to voiceless and vice-versa, and four constraints: *VOICE penalizes each voiced obstruent; *CODA-VOICE penalizes each voiced obstruent in a coda; *VTV penalizes voiceless consonants intervocalically; and IDENT-VOICE penalizes any form resulting from a voice-switching operation. The number of constraint violations is indicated underneath each constraint.

(1) *Step 1 of example typology calculation: Violation profiles*

/bakad/	*VOICE	*CODA-VOICE	IDENT-VOICE	*VTV
a. bakad	2	1		1
b. bakat	1		1	1
c. bagad	3	1	1	
d. pakad	1	1	1	1

OT-Help next determines which of the candidates are possible optima — i.e., which are not harmonically bounded. It does this with Recursive Constraint Demotion (RCD) (Tesar & Smolensky 1998). Each candidate is set to be the winner, and RCD is applied to the tableau. If RCD succeeds, then the candidate is a possible optimum; if it fails, then the candidate is harmonically bounded.

Applied to (1), RCD finds that [bakad] (1a), [bakat] (1b), and [bagad] (1c) are possible optima with this constraint set, but [pakad] (1d) is not. (It is harmonically bounded by [bakat].) Although the optima have now been determined, OT-Help is still far from finished. For each optimum found at Step 1, it is necessary to trace out its derivational futures until the point of convergence, when no further changes are possible. For this purpose, the ranking conditions that made a candidate an optimum at Step 1 are crucial, because a candidate's derivational future must be consistent with the rankings that determined its derivational past. For example, the tableau in (2) shows the ranking conditions required to make [bagad] optimal. This tableau is in comparative format (Prince 2002) with *W*s and *L*s indicating whether a constraint favors the winner or the loser. The ranking conditions derive from the requirement that any constraint preferring a loser must be dominated by some constraint that prefers a winner (in the same row). The possible derivational futures of [bagad] must be consistent with these ranking conditions. For instance, it could not change back into [bakad] at the next step of the derivation, because that would require ranking *VOICE above *VTV, which is inconsistent with the *VTV » *VOICE ranking required to make [bagad] beat [bakad] and [pakad] in (2).

(2) *Ranking conditions for optimal* [bagad]
*VTV » *VOICE, *CODA-VOICE, IDENT-VOICE

/bakad/	*VTV	*VOICE	*CODA-VOICE	IDENT-VOICE
a. bakad	1 **W**	2 **L**	1	**L**
b. bakat	1 **W**	1 **L**	**L**	1
☞ c. bagad		3	1	1
d. pakad	1 **W**	1 **L**	1	1

To ensure consistency across the steps of a derivation, OT-Help retains the ranking conditions from each step in the form shown in (2): an indicated optimum, and violation profiles for it and its competitors; it does not retain the ranking returned by RCD. This data structure is what Prince & Tesar (2004) call the *Support*. By applying RCD to the entire Support constructed in this fashion, OT-Help ensures that the ranking requirements established at earlier steps are respected later on.

For example, when the winning [bagad] in (2) is submitted to the operations and constraints, it yields the violation tableau (3):

(3) *Step 2 from* /bakad/ *via* [bagad]*: Violation profiles*

bagad	*VTV	*Voice	*Coda-Voice	Ident-Voice
a. bagad		3	1	
b. pagad		2	1	1
c. bakad	1	2	1	1
d. bagat		2		1

To determine whether, say, [bagat] (3d) is a possible optimum at this step, we designate it as such, add the resulting ranking data to the support, and apply RCD. That is, we now apply RCD to the data in (4) along with those in (2).

(4) *Ranking conditions for* [bagad] → [bagat]

bagad	*VTV	*Voice	*Coda-Voice	Ident-Voice
a. bagad		3 **W**	1 **W**	**L**
b. pagad		2	1 **W**	1
c. bakad	1 **W**	2	1 **W**	1
☞ d. bagat		2		1

RCD succeeds, so /bakad/ → [bagad] → [bagat] is a possible (partial) derivation.

OT-Help proceeds in this fashion, tracing out each possible derivation until convergence. At the end, each derivation has associated with it a Support containing ranking information for the optima at every step. The ranking that OT-Help reports is the ranking that RCD gives for this Support.

So far, the discussion has dealt with only a single initial input (underlying representation). When there are multiple initial inputs, OT-Help first determines the derivations that are possible with each one taken individually. To find out whether two derivations from different initial inputs can coexist in a language, all that is required is to combine the Supports for the two derivations and check whether RCD is successful. (This is an adaptation of the typology calculation method from OTSoft (Hayes et al. 2003)).

The process of determining a typology is identical in serial HG, except that OT-Help uses an application of linear programming (HaLP (Potts *et al.* 2010)) rather than RCD to

find optima, and it reports constraint weightings rather than rankings. A judgment that a particular system is infeasible has the same role in the serial HG calculation as does failure of RCD in the serial OT calculation.

In section 2, we show how to construct the operation and constraint files for this example and then how to use OT-Help to find the serial OT and HG typologies. First, though, we must provide a somewhat lengthy explanation of how OT-Help deals with ties.

1.2 Treatment of Ties

In serial OT and HG, ties in intermediate steps of a derivation can occur even when the final output is unique. Consider for example the initial input /bada/ with the ranking *VOICE » IDENT-VOICE. Given the limitation to a single application of the [voice] changing operation at each step of the derivation, the candidate set for the first step will be as shown in (5). The two candidates with devoicing have identical violation profiles and thus tie for optimality.[2]

(5) *Step 1 from* /bada/

/bada/	*VOICE	IDENT-VOICE
a. bada	2 **W**	
☞ b. pada	1	1 **L**
☞ c. bata	1	1 **L**

Given this ranking, the next step will devoice the remaining voiced stop in either of the two optima (5b) or (5c), thus producing [pata], which will emerge as the unique final output. Ties typically (though not invariably) arise as in this case when the same process is applicable at multiple loci in a form. It can only apply to one of them at each step, but eventually it affects all of them.

[2] For purposes of this illustration, we ignore the potential presence of any tie-breaking constraint, like *VTV, in the hierarchy. We also put aside the possibility of defining optimality as requiring a unique optimum, which would lead to a tableau like that in (5) having no optimum and, in the context of OT-Help, would presumably lead to a request to the user to supply a further constraint. The implementations of RCD and HaLP in OT-Help do in fact impose such a uniqueness requirement. This is circumvented in serial typology calculations by temporarily merging candidates with identical violations.

How should tied winners at intermediate steps be handled in serial OT and HG? There are at least three possibilities to consider: (a) all winners are treated as inputs to a single tableau; (b) one winner is chosen at random; or (c) the derivation forks into separate tableaux, one for each winner. We consider each in turn.

In the "multiple input" method, tied optima are taken as inputs to a single shared tableau in the next step of the derivation. The operations apply to each one to produce the candidate set, and candidates derived from each of the inputs compete with one another. Since we have multiple inputs in the step after (5), they are given in the appropriate rows of the tableau: (6a–c) have input [pada], and (6d–f) have input [bata].

(6) *Step 2 from* /bada/

		*Voice	Ident-Voice
	a. pada → bada	2 **W**	1
☞	b. pada → pata		1
	c. pada → pada	1 **W**	**L**
	d. bata → bada	2 **W**	1
☞	e. bata → pata		1
	f. bata → bata	1 **W**	**L**

The two input–output mappings leading to [pata] are chosen as optimal, and the derivation subsequently converges on the final output [pata].

In the "random choice" method, one member of any set of tied optima is randomly selected as the input for the next step of the derivation. Choosing either [pada] or [bata] as the input to Step 2 will lead to [pata] as its output (as can be confirmed by examining (6a–c) and (6d–f) respectively).

In the third "branching derivation" method, the derivation branches when there are multiple optima, with each one becoming an input to its own tableau. Under this approach, (6a–c) would be the tableau for Step 2 of one branch of the derivation, and (6d–f) would be the tableau for a separate branch. One would then require a decision rule to choose among the branches, though this could be as simple as accepting the output if all branches lead to the same outcome (as they do in our example), and rejecting it if they do not.

For the example above, all three of these possibilities would lead to derivations mapping initial /bada/ to final [pata]. In many cases of ties, the same single outcome is produced under all three approaches, and the choice amongst them would have no empirical consequence. Since there were no obvious differences amongst them, an early implementational decision was made to use the multiple input method in OT-Help. We have subsequently discovered that under certain circumstances the multiple input method can yield a unique optimum when the random choice and branching derivation methods yield multiple optima, and that these unique optima may conceal underlying problems in the constraint set (see the handouts by McCarthy and Pruitt mentioned at the end of the section).

1.2.1 *Input Death*

To produce an example of a tie from which the multiple input method yields a unique optimum while the others do not, we can modify our last example by adding a further operation that places stress on a syllable; note that candidates are limited to an application of either the voice-changing or stress-placing operation. We also adopt a different set of markedness constraints: OCP-Voice penalizes a word with two voiced obstruents, *StressedVoicedOnset penalizes a voiced onset of a stressed syllable, and Stress-Right penalizes any word that lacks stress on the rightmost syllable. The tableau in (7) shows that the outcome of the first step is the same as above, in that devoicing either obstruent is equally optimal. The final syllable remains unstressed because OCP-Voice ranks above Stress-Right.

(7) *Step 1 from* /bada/

/bada/	OCP-Voice	Ident-Voice	Stress-Right	*Stressed-VoicedOnset
a. bada	1 **W**	**L**	1	
☞ b. pada		1	1	
☞ c. bata		1	1	
d. badá	1 **W**	**L**	**L**	1 **W**
e. báda	1 **W**	**L**	1	1 **W**

The second step using the multiple input method is shown in (8). To save space, we only consider candidates with devoicing (and not voicing), and with stress placement on the final syllable (and not on the initial one). The noteworthy aspect of this tableau is that only one of the inputs has an optimal mapping from it: [pada] → [padá] loses because the onset of its stressed syllable is voiced, in contrast with optimal [bata] → [batá]. We call a situation in which one of the inputs does not have an optimal mapping from it an "input death".

(8) *Step 2 from* /bada/

		OCP-Voice	Id-Voice	Stress-R	*Stressed-VoicedOnset
	a. pada → pada			1 W	
	b. pada → pata		1 W	1 W	
	c. pada → padá				1 W
	d. bata → bata			1 W	
	e. bata → pata		1 W	1 W	
☞	f. bata → batá				

On the next step, the derivation will converge on the final output [batá].

Consider now what would happen if we chose randomly between the optima in (7), or constructed a branching derivation from each of them. In both cases, [pada] → [padá] would no longer compete with [bata] → [batá], and would be optimal in its own tableau. The random choice method would yield variation between final outputs [padá] and [batá], while the branching derivation would have [padá] and [batá] as the final outputs of its two branches. Just as tied optima have not turned out to be a good theory of variation in parallel OT (McCarthy 2002: 200), multiple final outputs resulting from intermediate ties are unlikely to form the basis of a reasonable theory of variation in serial OT or HG. Instead, they are likely always an indication that the constraint set needs work because it is failing to make a crucial distinction (see again the McCarthy and Pruitt handouts for examples).

By producing a unique final output when the other methods yield multiple final outputs, the multiple input method used in OT-Help can sometimes hide problems in the constraint set. Therefore, all languages that contain an input death of the type illustrated

in (8) are indicated in the typology display with an asterisk, and the input deaths are marked with daggers in the derivation displays. A derivation with an input death will usually (but not always) yield multiple optima under the other approaches to ties. Users encountering these warnings should proceed cautiously in interpreting their results and should consider introducing an additional constraint (even an ad hoc one) to break the intermediate tie.

For further discussion of ties, see the handouts by John McCarthy and Kathryn Pruitt, which are available as the following PDFs:

- http://web.linguist.umass.edu/~mccarthy-pater-nsf/Knotty-ties.pdf
- http://web.linguist.umass.edu/~mccarthy-pater-nsf/Ties-in-HS.pdf
- http://web.linguist.umass.edu/~mccarthy-pater-nsf/Remark-on-ties-in-HS.pdf
- http://web.linguist.umass.edu/~mccarthy-pater-nsf/Further-thoughts-on-ties.pdf

2 Using OT-Help 2.0

This section shows how OT-Help displays serial typologies and describes how to create the simple serial factorial typology examples in section 1.

There are three typologies: (a) the simple voicing typology of section 1.1, (b) the typology with tied derivational paths and multiple outputs (section 1.2), and (c) the typology with tied derivational paths and input death (section 1.2.1). Each typology has three user files, so there are a total of nine files used in this section. These files can be directly downloaded in `.ZIP` format from http://web.linguist.umass.edu/~OTHelp/OTHexs.zip. All nine files occur in the directory named `Section2`. (The `ZIP` file also contains further operation and constraint examples in separate directories.)

Here we are not concerned with parallel typologies and associated issues.

The user guide contains special information boxes to help guide the reader. There are three such boxes. *Technical Note* boxes explain esoteric topics or detailed tangents pertaining to OT-Help for advanced users — beginning users should feel free to skip over

these boxes. *Helpful Hint* boxes contain advice, warnings, or further information for beginning users.

2.1 Platforms and System Requirements

OT-Help works on Windows, Mac, and Linux systems.

OT-Help requires Java version 5 or later.

2.2 Program Files

OT-Help consists of two files: the main OT-Help Java archive file (`.JAR`) and the linear programming solver `JAR` file developed by OpsResearch.com. The main file is available for download from

http://web.linguist.umass.edu/~OTHelp/OT-Help2.jar

and the linear programming solver can be downloaded from

http://opsresearch.com/cgi-bin/freeware.cgi/or124.jar

OT-Help is a standalone program that is ready to run without running a separate installer program; however, a minimal installation step is required for the `or124.jar` file.

2.3 Installation

The linear programming `or124.jar` file must reside in a subdirectory named `OTH-lib`, which must be nested within the same directory containing the `OT-Help2.jar` file. You must create this `OTH-lib` directory yourself.

2.4 Starting the Program

OT-Help is run via the Windows, Mac, or Linux GUI.

2.5 User Files

When calculating a serial typology, three user files are needed:

1) input file <*filename*>.txt
2) operation file <*filename*>.txt_OPERATIONS
3) constraint file <*filename*>.txt_CONSTRAINTS

The user files utilized in this section and throughout the user guide (which you extracted from OTHexs.zip[3]) are

bakad.txt
bakad.txt_OPERATIONS
bakad.txt_CONSTRAINTS

The input file contains the input candidates, the operation file contains definitions of the possible modifications that the input can undergo, and the constraint file contains definitions of violable constraints.

In the following, we describe some general details about user files, and then more specific details about each of the file types in subsections using the typology example of section 1.

File Names & Extensions. In order for OT-Help 2.0 to work properly, the files must have the correct file extensions shown above. Additionally, the filename for all three files must be the same.[4]

File Format. All user files must be in tab-delimited plain text (.TXT).

[3] This file is available from http://web.linguist.umass.edu/~OTHelp/OTHexs.zip.
[4] Filenames and file extensions are not case-sensitive.

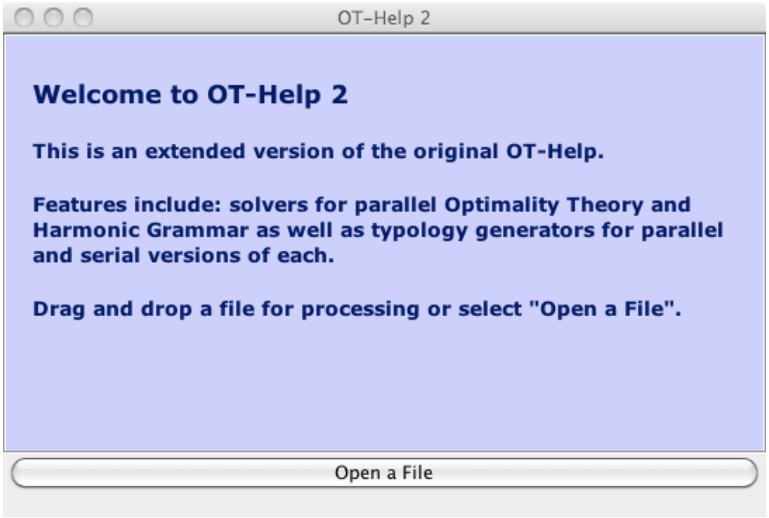

OT-Help 2
Welcome to OT-Help 2
This is an extended version of the original OT-Help.
Features include: solvers for parallel Optimality Theory and Harmonic Grammar as well as typology generators for parallel and serial versions of each.
Drag and drop a file for processing or select "Open a File".
Open a File

Fig. 1: Initial OT-Help 2.0 window.

Technical Note

Starting via the Console

OT-Help can be run through the console with the following command:

```
java -jar OT-Help2.jar
```

(The console on Windows is the DOS Command Prompt. It is the Terminal on Macs. If your current directory is not the same as the directory that `OT-Help2.jar` is in, then you will need to provide the directory path.)

The benefits of running OT-Help through a console are that processing and errors will be displayed. Currently, the OT-Help GUI does not show a graphical difference between processing a solution and not processing, which means that in some cases it isn't clear whether the program has crashed or if it is still processing. When OT-Help is run through a console, the current state of processing is displayed in a console in real time. Additionally, if there are, for example, formatting errors in your input file, then the console will display the exceptions and error codes.

HELPFUL HINT

Since the user files are text files, of course, simple text editors may be used. If you use a word processor (e.g., Microsoft Word, Apple's Pages, OpenOffice Writer), you must make sure to save it in the plain text format. Since all user files are tab-delimited, it may be useful to edit them using a spreadsheet program (e.g., Microsoft Excel, Apple Numbers, OpenOffice Calc) making sure to save them in the *Tab Delimited Text* format.

Technical Note

Special Characters and Character Encoding

Since the OT-Help GUI uses HTML, HTML character entities can be used for non-ASCII characters — for example, `á` for the lowercase *a* with an acute accent < á > and `ɒ` or `ɒ` for the rounded low back vowel < ɒ >. Although OT-Help can read text files with UTF-8 character encoding, non-ASCII characters will not display correctly, so character entities must be used. Plain text files in UTF-16 are not readable. (Note: it might generally be easier to use schematic data with only ASCII characters rather than dealing with actual words with sequences of character entities for Unicode phonetic transcription.)

2.5.1 Input File Internal Format

The main components of the tab-delimited input file are a two-line preamble, the list of inputs, and the end of file marker `[end of tableaux]`. The format of the input file is partly due to its legacy compatibility with OTSoft and OT-Help 1.2.[5]

In the `bakad.txt` file (Fig. 2), there are two inputs: `bakad` and `bada`.

The First Two Lines. The first line of the file should contain a `[typology]` tag, and the second line should contain a `[begin tableaux]` tag.

[5] For information about the OTSoft file format, see the OT-Help 1.2 user guide available from http://web.linguist.umass.edu/~OTHelp/OTHelp.pdf or the OTSoft user manual available from http://www.linguistics.ucla.edu/people/hayes/otsoft/OTSoftManual.pdf.

```
[typology]
[begin tableaux]
bakad            0  1
bada             0  1

[end of tableaux]
```

Fig. 2: Contents of input file `bakad.txt`.

HELPFUL HINT

In this user guide all user file examples have their columns lined up, but when using a text editor or word processor, this may not be the case depending on the length of the strings in each column and the positions of the tabs in your editor/processor. In other words, you should resist the temptation to line the columns up by adding more than one tab. Adding extra tabs or forgetting to add tabs where needed will break OT-Help's predictable behavior. (Of course, if you use a spreadsheet to create your files, then the columns will line up.)

The Following Input Rows. After the two-line preamble is the list of inputs, which is tab-delimited into three columns. Only a *single tab* separates the columns. The first column contains your desired inputs. The second column can contain any value.[6] The values are ignored in serial typology calculation. Here we have used `0` as a convenient placeholder. The third column should contain `1`.[7]

Whitespace. Adding whitespace (tabs, linebreaks) after the third column is permissible. For example, in the input file `bakad.txt` here (Fig. 2), we have chosen to skip a line between the last input row and the `[end of tableaux]` marker. You may wish to add whitespace as needed for human readability.

[6] An alphanumeric character is required in this column. Otherwise, the entire input row is ignored.

[7] Actually, having a `1` here is not strictly necessary for serial typologies. If a different alphanumeric character is present here, OT-Help will just display the message *Some actions are not available, because your data does not specify a single winner in each tableau* and the parallel HG solution and parallel OT solution options in the home page window will not be available (which is irrelevant to serial typology calculation).

Technical Note

Serial & Parallel Typology Calculation

In this section, we have assumed that only serial typologies are desired. However, since OT-Help also does parallel OT and HG typology calculation and uses the OTSoft file format, the same input files can be used for both parallel and serial typologies. In this case, when a serial typology is computed from a file that is also used for parallel typologies, information about parallel output candidates and their violation profiles will just be ignored as the serial typology generates outputs based on user-defined operations and assigns violations according to user-defined constraints.

2.5.2 Operation File

The operation file defines the operations or changes that can be made on the input strings specified in the input file. These changed strings *in addition to* the faithful candidate are present in tableaux and evaluated using any user-defined constraints. This subsection will only introduce operations using the simple typology calculation discussed in section 1.

A full description of the operation file format is in section 3.

```
[operation]
[long name]       ChangeVoicing
[active]          yes
[definition]      p              b
[definition]      b              p
[definition]      t              d
[definition]      d              t
[definition]      k              g
[definition]      g              k
[violated faith]  Ident-Voice

[end operations]
```

Fig. 3: Contents of operation file `bakad.txt_OPERATIONS`.

Our example file `bakad.txt_OPERATIONS` is shown in Fig. 3. This file defines a single operation, which we have named `ChangeVoicing`. Our operation has a six-line definition. In the definition part of the operation, the substring that undergoes a change appears in the middle column while substring that it changes into appears in the rightmost column. Our operation example replaces a `p` input character with a `b` character and vice versa, a `t` input character with a `d` character and vice versa, and a `k` input character with a `g` character and vice versa. The formulation here is context free, so the input string `bakad` in the input file `bakad.txt` (Fig. 2) would have three possible one-step changes (`b` → `p`, `k` → `g`, `d` → `t`) under our `ChangeVoicing` operation while the second input string `bada` would have two possible changes (`b` → `p`, `d` → `t`):

`bakad`	→	`pakad`
		`bagad`
		`bakat`
			`bada`	→	`pada`
					`bata`

Operation definitions may also include regular expressions and can be written as context-dependent changes.

All of these strings modified by our `ChangeVoicing` operation are output candidates — along with the faithful candidates — which are then evaluated for constraint violation and may become inputs to the next step in the derivation. Our example has the following input–output correspondences:

Input	Output Candidates	Input	Output Candidates
`bakad`	`bakad`	`bada`	`bada`
	`pakad`		`pada`
	`bagad`		`bata`
	`bakat`		

These input–output correspondences were discussed in section 1 in tableaux (1) and (5), respectively.

Operations & Faithfulness. Any operation may result in a change that violates one or more faithfulness constraints. Because of this tight connection between operation and

violation of faithfulness, operations indicate which faithfulness constraint they violate through the [violated faith] parameter(s). Thus, faithfulness constraints are actually defined within the operation file (unlike markedness constraints). In our example (Fig. 3), any of the changes under the ChangeVoice operation will incur a violation of a constraint named Ident-Voice. Thus, the changed strings (pakad, bagad, bakat; pada, bata) will each received one violation mark from the Ident-Voice constraint. Additionally, in order for faithfulness constraints to be active in typology calculations, they must be indicated as active in the constraint file.

See section 3 for further information on operations.

2.5.3 Constraint File

The constraint file defines constraints and determines whether these constraints are in use in serial typologies. This subsection will only introduce four constraints using the simple typology calculation discussed in section 1. A full description of the constraint file format is in section 4.

Our example file bakad.txt_CONSTRAINTS is shown in Fig. 4. This file includes four constraints *Voice, *Coda-Voice, *VTV, and Ident-Voice, which correspond to the constraints of the same names in section 1 (introduced on page 4). Here we consider the violations assigned by these constraints to the outputs at the first step of the derivations from inputs /bakad/ and /bada/ as shown in tableaux (1) and (5).

Markedness Constraint Examples. *Voice, *Coda-Voice, and *VTV are all markedness constraints as indicated by their [type] of markedness. Since they are markedness constraints, they include a definition parameter. Markedness constraints are essentially substring matching searches and are defined using regular expressions: an output string receives a violation for each match found by the regular expression search.

The regular expression definition of *Voice is [bdg], which will match every instance of b, d, or g in an output string. Although the real *Voice penalizes all voiced obstruents,

```
[constraint]
[long name]   *Voice
[active]      yes
[type]        markedness
[definition]  [bdg]

[constraint]
[long name]   *Coda-Voice
[active]      yes
[type]        markedness
[definition]  [bdg]$

[constraint]
[long name]   *VTV
[active]      yes
[type]        markedness
[definition]  a[ptk]a

[constraint]
[long name]   *Ident-Voice
[active]      yes
[type]        faithfulness

[end constraints]
```

Fig. 4: Contents of constraint file `bakad.txt_CONSTRAINTS`.

our schematic example only considers these three stops, and so we only need to incorporate these into our regular expression. Given the input string of `bakad` from section 2.5.1 and the operation `ChangeVoicing` from section 2.5.2, the output candidates at the first step of the derivation are `bakad`, `pakad`, `bagad`, and `bakat` as discussed in section 1. The outputs `bada`, `pada`, and `bata` are candidates at the first step of the derivation from our second input `bada`. The `*Voice` constraint will find the following matches:

Candidate String	Number of Matches	Substring Match(es)
bakad	2	b , d
pakad	1	d
bagad	3	b , g , d
bakat	1	b
bada	2	b , d
pada	1	d
bata	1	b

The number of matches found by *Voice is the number of violations it doles out to each of these candidates.

*Coda- Voice (Fig. 4) is a context-dependent version of *Coda and, accordingly, its regular expression is more specific. In section 1, codas were not explicitly indicated and were assumed to occur in the five-segment forms (e.g., the [d] in [bakad]) and not to occur in four-segment forms (e.g., [bada]). Following this, we can choose to implement a coda position as the end of line context which can be referenced by the regular expression metacharacter $. Our *Coda- Voice definition of [bdg] $ will then match any instance of b, d, or g that occurs at the end of a string. Thus, *Coda- Voice will find the following matches for our same two sets of output candidates:

Candidate String	Number of Matches	Substring Match
bakad	1	d
pakad	1	d
bagad	1	d
bakat	0	
bada	0	
pada	0	
bata	0	

Our last markedness constraint is *VTV (Fig. 4), which prefers intervocalic voicing. Its regular expression definition a[ptk] a will match all instances of p, t, or k flanked by a on both sides. (And, we just use a to represent schematically any vowel in our regular expression.) Thus, our first-step outputs receive the following violations:

Candidate String	Number of Matches	Substring Match
bakad	1	aka
pakad	1	aka
bagad	0	
bakat	1	aka
bada	0	
pada	0	
bata	1	ata

Faithfulness Constraint Example. Our example faithfulness constraint is `Ident-Voice` (Fig. 4), and our single operation example is `ChangeVoicing`, which is found in our operation file (Fig. 3). Since the argument of `ChangeVoicing`'s `[violated faith]` parameter is `Ident-Voice`, every candidate that is generated via the `ChangeVoicing` operation will receive a violation of `Ident-Voice`. In order to activate the constraint so that it appears in tableaux, `Ident-Voice` is also listed in the constraint file. `Ident-Voice`'s `[active]` parameter is set to `yes`, its `[type]` is `faithfulness`.

Our input strings in this example were `bakad` and `bada`. At the first step of their derivations, the output candidates `pakad`, `bagad`, & `bakat` and `pada` & `bata` are generated via `ChangeVoicing`, so these candidates violate `Ident-Voice`. The other candidates `bakad` and `bada` are not generated by any operation but are rather identical to their inputs, and so they do not violate `Ident-Voice`.

Input	Output Candidate	Violated `Ident-Voice`?
bakad	bakad	
bakad	pakad	yes
bakad	bagad	yes
bakad	bakat	yes
bada	bada	
bada	pada	yes
bada	bata	yes

See section 4 for further information on constraints.

2.5.4 Commenting

There are two ways to add comments to user-files. First, all user files allow for comment text to occur after the end of file markers. Any text after these markers is ignored by OT-Help. For reference, the three end of file markers are the following:

input file	`[end of tableaux]`
operation file	`[end operations]`
constraint file	`[end constraints]`

Comments can be added to operation and constraint files via a `[comment]` tag. This is useful since placing all comments after the end of file marker can be inconvenient if the operation or constraint file contains several operations/constraints. The format is similar to the operation/constraint parameters: the `[comment]` tag is separated from the comment text by a single tab. The following example (Fig. 5) places a multi-line comment about *Coda-Voice between two constraints.

2.6 Loading User Files

Files can be loaded into OT-Help via two methods: drag-&-dropping and using the *Open a File* button.

Drag-&-Drop. This drag-&-drop method is available for Windows and Mac systems but not for Linux. All three user files can be loaded into OT-Help by drag-&-dropping the input file into the OT-Help GUI window. Doing so will automatically load the operation and constraint files. Operation and constraint files cannot be drag-&-dropped into OT-Help. An input file can be drag-&-dropped onto any window.[8]

***Open a File* Button.** This method is available for Windows, Mac, and Linux systems. As with the drag-&-drop method, you can only open an input file. The associated operation and constraint files are automatically loaded with the input file. The *Open a File* button is available on all OT-Help windows.

[8] There is a reminder to this effect on most windows: *To process a new file, drop it in this window or click "Open a File".*

```
[constraint]
[long name]   *Coda-Voice
[active]      yes
[type]        markedness
[definition]  [bdg]$

[comment]     *CODA-VOICE penalizes voiced consonants
[comment]     in coda position.
[comment]     Its regex finds all voiced stops (b,d,g)
[comment]     at the end of the line ($).

[constraint]
[long name]   *VTV
[active]      yes
[type]        markedness
[definition]  a[ptk]a

[end constraints]
```

Fig. 5: Modified constraint file with four comment lines.

Helpful Hint

Commenting you user files is generally a good practice. Since regular expressions can be tricky to create and understand, adding comments about how they work can save you and others time. Also, you might want to add comments about operations/constraints that you were using initially but discarded later in favor of other operations/constraints.

Refreshing File Changes. If changes are made to the input, operation, and/or constraint files, then the input file must be re-loaded (using either method) into OT-Help in order for the changes to take effect.

HELPFUL HINT

OT-Help currently has no error message feedback for users. If the constraint or operation file is not found, the OT-Help GUI will not display an error. If the constraint file is missing, the typology will be calculated with no constraints. If the operation file is missing, the typology will not be calculated — OT-Help will appear to do nothing.

2.7 Home Page Window

After the three files are loaded, OT-Help displays a window with the name of the input file loaded and a list of actions. This window is referred to as the *home page*. The home page window screenshot shown in Fig. 6 shows that the input file `bakad.txt` with path `/Applications/OTHelp/` was loaded.

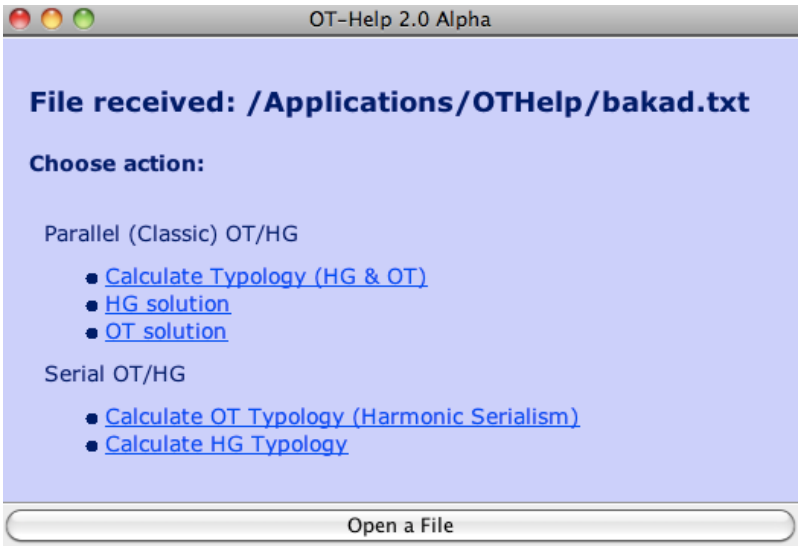


Fig. 6: Home page window.

Action Links. Each action is an underlined blue link. The serial typology calculation links — *Calculate OT Typology (Harmonic Serialism)* and *Calculate HG Typology* — are

directly below the parallel typology calculation links. Clicking a link will take you to the appropriate typology window.

Unique Languages Option. After choosing either the OT or HG option, an option popup window appears with a check box for displaying languages with a unique set of outputs.[9] A screenshot of this appears in Fig. 7.

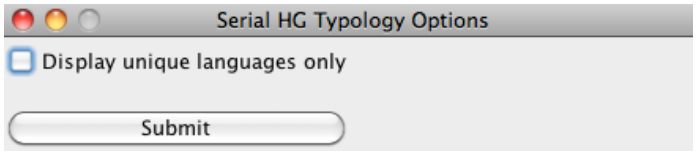


Fig. 7: Unique languages option popup.

The *Display unique languages only* checkbox conflates all languages that may differ in constraint ranking/weighting (and also possibly derivational path) but result in the same set of surface outputs.

The default behavior (with the box unchecked) is to display all languages.

After choosing whether to check the box, OT-Help will process the results in a new language typology window.

2.8 Language Typology Window

The typology window displays the total number of languages found and optionally the number of unique language if this differs from the total number. The number of languages is followed by a table in which each row is a language and each column is the output in that language for a given input and a list of constraint rankings/weightings for each grammar.

We show the typology windows for the OT and HG typologies of our simple example from section 1.

Serial OT Typology Window. The typology window for serial OT using the user-defined files discussed in sections 2.5.1, 2.5.2, and 2.5.3 is shown in Fig. 8.

[9] If this popup window is closed, then OT-Help will also be closed.

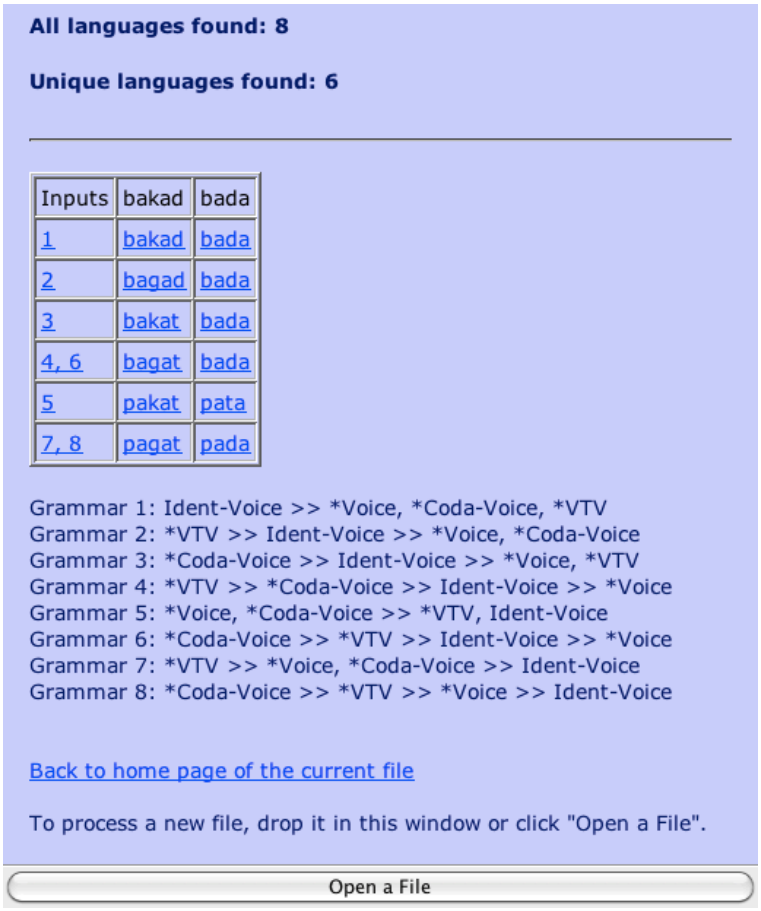

All languages found: 8
Unique languages found: 6
Inputs bakad bada
1 bakad bada
2 bagad bada
3 bakat bada
4, 6 bagat bada
5 pakat pata
7, 8 pagat pada
Grammar 1: Ident-Voice >> *Voice, *Coda-Voice, *VTV
Grammar 2: *VTV >> Ident-Voice >> *Voice, *Coda-Voice
Grammar 3: *Coda-Voice >> Ident-Voice >> *Voice, *VTV
Grammar 4: *VTV >> *Coda-Voice >> Ident-Voice >> *Voice
Grammar 5: *Voice, *Coda-Voice >> *VTV, Ident-Voice
Grammar 6: *Coda-Voice >> *VTV >> Ident-Voice >> *Voice
Grammar 7: *VTV >> *Voice, *Coda-Voice >> Ident-Voice
Grammar 8: *Coda-Voice >> *VTV >> *Voice >> Ident-Voice
Back to home page of the current file
To process a new file, drop it in this window or click "Open a File".
Open a File

Fig. 8: Serial OT typology window with unique languages option selected.

Here OT-Help calculated eight languages of which two pairs result in the same set of outputs. In other words, there are six unique languages. The unique languages information is optional and accessible through the *Display unique languages only* popup window (Fig. 7), which pops up after the home page window (section 2.7). Fig. 8 shows the OT typology with the unique languages option selected.

In the table, the first row indicates the inputs, which here are `bakad` and `bada`. Beneath the first row are the six rows of unique languages labelled numerically in the first column. In the subsequent columns are the outputs that correspond to the inputs in the first row

of the same column. For example, Language 2 (Fig. 8) has input bakad mapping to output bagad and input bada mapping to output bada.

Since the unique languages options has been selected here, different languages that generate the same set of input–output mappings have their language numbers appear in the same table row. For example, both Language 4 and Language 6 (Fig. 8) have intervocalic voicing and coda devoicing of input /bakad/ and a faithful surfacing of input /bada/.

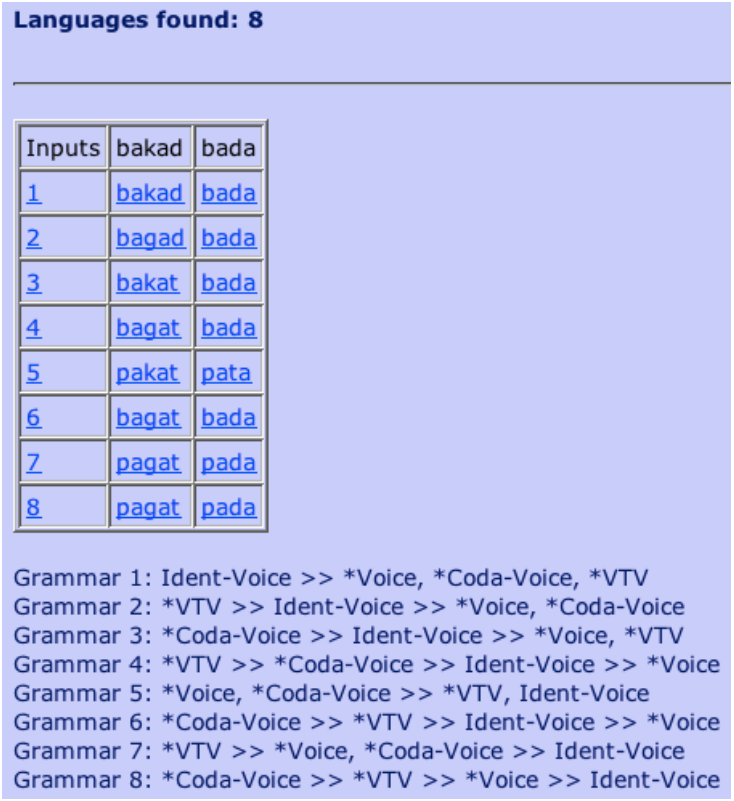
After the table is the list of constraint rankings that comprise each grammar. The numbers in the table correspond to the numbers in the constraint ranking list. For example, the Language 2 (Fig. 8) that produced the /bakad/–[bagad] and /bada/–[bada] mappings has a constraint ranking of *VTV » Ident-Voice » *Voice, *Coda-Voice. Also, even though Languages 4 and 6 (Fig. 8) have the same input–output mappings, we see that their constraint rankings differ.

Derivation Window Links. The typology window (Figs. 8, 9, 10) also has underlined blue links that navigate the user to other windows. Within the table, the numbers in the first column corresponding to each language are linked to a short derivation window for that language. Additionally, each output string is a link to a short derivation for only that particular input–output derivation. Derivation windows are explained in section 2.9.

Return to Home Page Link. The last link *Back to home page of the current file* (shown in Fig. 8) is a return link to the home page window discussed in section 2.7. This return link is available at the bottom of all language typology and derivation windows.

Display Without *Unique Languages* Option. If the unique languages option checkbox was not selected after the home page window, then constraint ranking's input–output mapping set is displayed on a separate table row. The same serial OT typology window appears as in Fig. 9 but with the unique language display option not selected.

Serial HG Typology Window. The typology window of serial HG typologies is largely the same as serial OT typology windows. The corresponding HG typology is shown in Fig. 10. Here we have again selected the unique languages option.



Languages found: 8

Inputs	bakad	bada
1	bakad	bada
2	bagad	bada
3	bakat	bada
4	bagat	bada
5	pakat	pata
6	bagat	bada
7	pagat	pada
8	pagat	pada

Grammar 1: Ident-Voice >> *Voice, *Coda-Voice, *VTV
Grammar 2: *VTV >> Ident-Voice >> *Voice, *Coda-Voice
Grammar 3: *Coda-Voice >> Ident-Voice >> *Voice, *VTV
Grammar 4: *VTV >> *Coda-Voice >> Ident-Voice >> *Voice
Grammar 5: *Voice, *Coda-Voice >> *VTV, Ident-Voice
Grammar 6: *Coda-Voice >> *VTV >> Ident-Voice >> *Voice
Grammar 7: *VTV >> *Voice, *Coda-Voice >> Ident-Voice
Grammar 8: *Coda-Voice >> *VTV >> *Voice >> Ident-Voice

Fig. 9: The same serial OT typology window without unique languages option.

In the list of HG grammars which is shown below the input–output table, the constraints are grouped according to weight (in parentheses) in descending order of weight. Each weight group is separated with a greater than sign (>). For example, HG Language 2 (Fig. 10) has `*VTV` with a weight of 5.0, `Ident-Voice` with a weight of 3.0, and `*Voice` and `*Coda-Voice` each with a weight of 1.0.

2.9 Derivation Windows

OT-Help has both *short derivation* and *full derivation* windows. Short derivations show only the winning output candidates at each step of a derivation. Full derivations show all competing outputs in tableaux for each step of a derivation. There are separate derivation

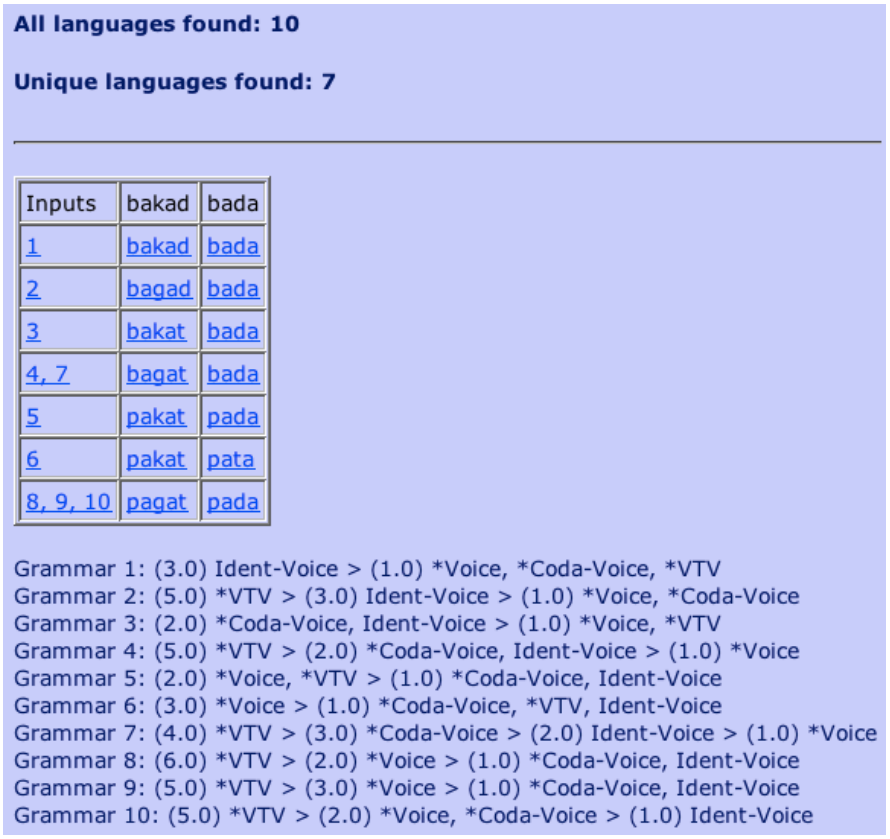

All languages found: 10
Unique languages found: 7
Inputs bakad bada
1 bakad bada
2 bagad bada
3 bakat bada
4, 7 bagat bada
5 pakat pada
6 pakat pata
8, 9, 10 pagat pada
Grammar 1: (3.0) Ident-Voice > (1.0) *Voice, *Coda-Voice, *VTV
Grammar 2: (5.0) *VTV > (3.0) Ident-Voice > (1.0) *Voice, *Coda-Voice
Grammar 3: (2.0) *Coda-Voice, Ident-Voice > (1.0) *Voice, *VTV
Grammar 4: (5.0) *VTV > (2.0) *Coda-Voice, Ident-Voice > (1.0) *Voice
Grammar 5: (2.0) *Voice, *VTV > (1.0) *Coda-Voice, Ident-Voice
Grammar 6: (3.0) *Voice > (1.0) *Coda-Voice, *VTV, Ident-Voice
Grammar 7: (4.0) *VTV > (3.0) *Coda-Voice > (2.0) Ident-Voice > (1.0) *Voice
Grammar 8: (6.0) *VTV > (2.0) *Voice > (1.0) *Coda-Voice, Ident-Voice
Grammar 9: (5.0) *VTV > (3.0) *Voice > (1.0) *Coda-Voice, Ident-Voice
Grammar 10: (5.0) *VTV > (2.0) *Voice, *Coda-Voice > (1.0) Ident-Voice

Fig. 10: Serial HG typology window with unique languages option selected.

windows for entire languages (i.e., the set of all input–output mappings) and individual derivation windows for only a single input–output pair in a given language.

Short Derivations. The short derivation windows are linked to directly from the typology window. Within the typology window, clicking on the language number links in the first column of the table navigates the user to a short derivation window for all the input–output mappings of that particular language.

In the typology window of our OT typology example (Figs. 8, 9), the link of Language 4 with outputs [bagat, bada] leads to the short derivation window for that serial OT language as shown in (Fig. 11).

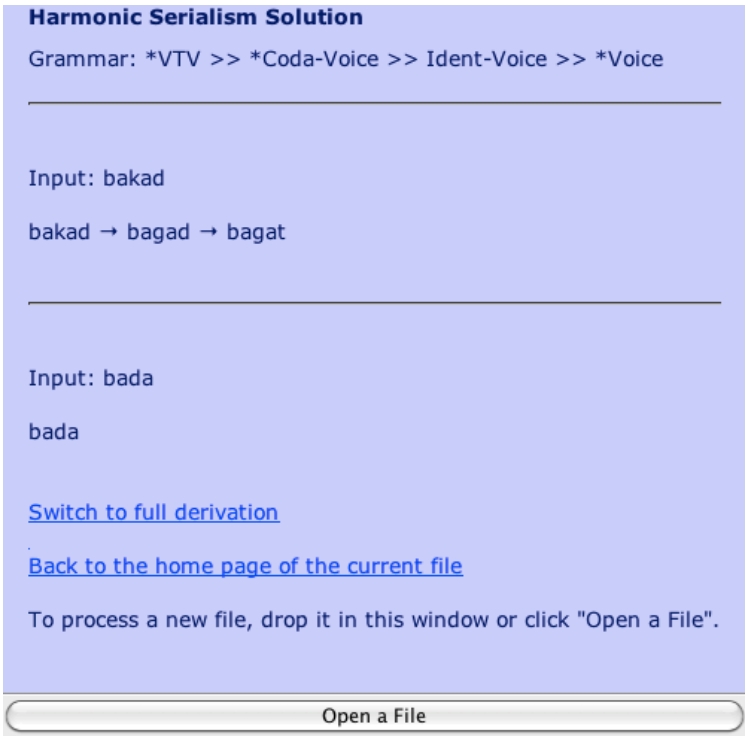

Harmonic Serialism Solution
Grammar: *VTV >> *Coda-Voice >> Ident-Voice >> *Voice
Input: bakad
bakad → bagad → bagat
Input: bada
bada
Switch to full derivation
Back to the home page of the current file
To process a new file, drop it in this window or click "Open a File".
Open a File

Fig. 11: Short derivation window for serial OT Language 4.

The constraint ranking for the language appears at the top of the short derivation window followed by each derivation from input to output. After each input, the derivation is presented in steps from the input form through intermediate forms (if any) to the final output form. The first input of Language 4 /bakad/ has a derivation of /bakad/ → [bagad] → [bakat] → [bakat]. In the short derivation window, the short derivation starts with the input `bakad` and then adds the subsequent forms in the short derivation `bagad` and `bagat` until the derivation converges:

`bakad` → `bagad` → `bagat`

The repetition of the last form under convergence (that is, … [bakat] → [bakat]) is not shown since this is predictable. The second input /bada/ has a shorter derivation of /bada/ → [bada], in which the output is completely faithfulful. Here the OT-Help derivation is displayed as just the input form `bada`.

At the bottom of each short derivation window, there is a link *Switch to full derivation* that navigates the user to the full derivation window (in addition to the home page link).

Helpful Hint

Navigation note: There is no way to go from a derivation window back to the typology window. You can only return to the home page window, and then you must recalculate the typology in order to get back to the typology window. (This limitation may change in a future version of OT-Help.)

However, you can run multiple instances of OT-Help at the same time. This may help you compare different derivations. Additionally, you could exploit this feature to compare a parallel typology with a serial typology or an OT typology with an HG typology.

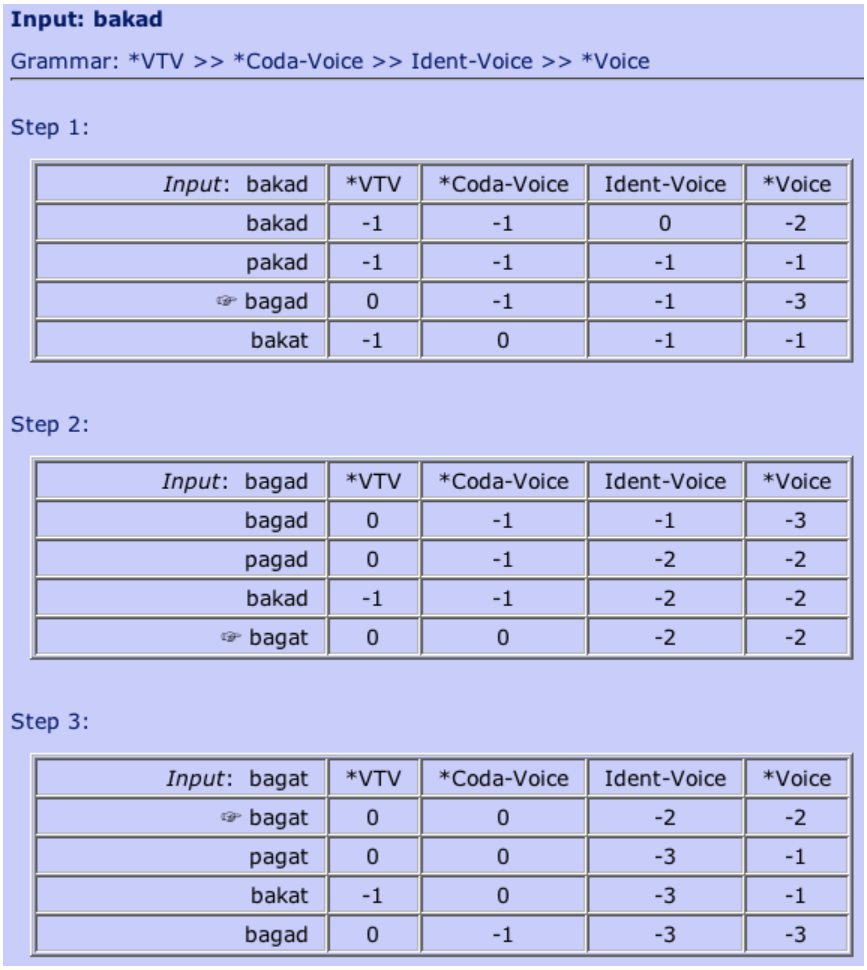
The serial HG short derivation window is identical to the OT short derivation window except that the language displays its constraint weightings.

Full Derivations. *Full derivation* windows show all candidates at each step in a derivation in tableaux. Full derivation windows are linked to from short derivation windows.

In (Fig. 12), the full derivation for for serial OT Language 4 is shown. We only show the full derivation for the first input /bakad/ due to the large window size. Inputs followed by the language's constraint ranking appear at the top of the window. After this is the series of tableaux — one for each derivational step — that displays all losing candidates and the optimum along with their constraint violation profiles.

At the bottom of full derivation windows is a *Switch to short derivation* link (not shown here due to space) that will navigate the user back to the short derivation window for that language.

Individual Derivations. Although all input–output derivations for each language can be presented in a short or full derivation window, OT-Help also has separate individual derivation windows for each of the input–output mappings. These windows may used when the user is not concerned with viewing all derivations for a language.



Input: bakad

Grammar: *VTV >> *Coda-Voice >> Ident-Voice >> *Voice

Step 1:

Input: bakad	*VTV	*Coda-Voice	Ident-Voice	*Voice
bakad	-1	-1	0	-2
pakad	-1	-1	-1	-1
☞ bagad	0	-1	-1	-3
bakat	-1	0	-1	-1

Step 2:

Input: bagad	*VTV	*Coda-Voice	Ident-Voice	*Voice
bagad	0	-1	-1	-3
pagad	0	-1	-2	-2
bakad	-1	-1	-2	-2
☞ bagat	0	0	-2	-2

Step 3:

Input: bagat	*VTV	*Coda-Voice	Ident-Voice	*Voice
☞ bagat	0	0	-2	-2
pagat	0	0	-3	-1
bakat	-1	0	-3	-1
bagad	0	-1	-3	-3

Fig. 12: A full derivation window for an input of serial OT Language 4.

All individual short derivation windows are linked to from the typology window (Figs. 8, 9, 10). Within the language table, every output is a link to an individual short derivation for that output.

Within the individual short derivation window is the same *Switch to full derivation* link that brings the user to an associated individual full derivation window.

Technical Note

Faithfulness Violations

OT-Help currently assigns faithfulness violations to output candidates that are derived from the input at each derivational step via a user-defined operation. Additionally, at each step of the derivation, all outputs candidates (including the faithful candidate) have one violation added to each violation profile for that faithfulness constraint. For example, at the step 1, a faithful candidate receives 0 violations while another unfaithful candidate receives 1 violation. At the step 2, the faithful candidate receives 0 + 1 violations while unfaithful candidate receives 1 + 1 violations. At the third step, faithful candidate receive 0 + 2, unfaithful receives 1 + 2. And so on.

2.10 Displaying Derivational Ties

2.10.1 Multiple Outputs

First, we consider how OT-Help displays a derivational tie that results in multiple identical outputs. In order to demonstrate derivational ties that were discussed in section 1.2, we remove the tie-breaking constraint `*VTV` from our `bakad` constraint set. For simplicity, we only consider the input `bada`. We have edited our original user files accordingly and saved them as the following three user files:

```
badaTies.txt
badaTies.txt_OPERATIONS
badaTies.txt_CONSTRAINTS
```

The `badaTies` input file only contains input `bada`. The `badaTies` operation file is identical to the original `bakad.txt_OPERATIONS` file. The `badaTies` constraint file is the same as the original `bakad.txt_CONSTRAINTS` file except with the `*VTV` constraint removed.

The OT typology generated from `badaTies` user files is shown in the typology window in Fig. 13.

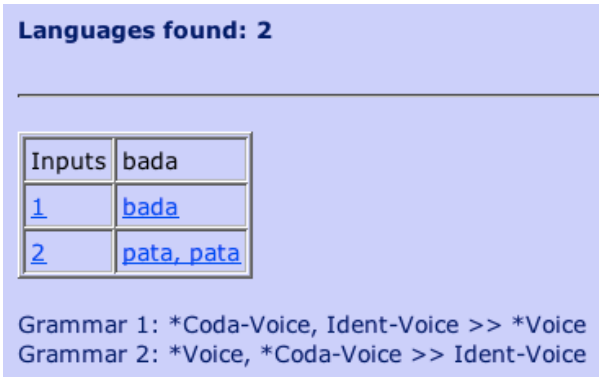


Fig. 13: Serial OT typology window with a language with multiple output optima.

In Language 2 of this typology, input /bada/ is mapped to two instances of the output [pata]. Essentially, this language devoices all stops that occur in the input (i.e., voicing is not contrastive). However, we have two derivational paths that lead to voicelessness since there are ties in the intermediate steps of the derivation from /bada/ to [pata]. This derivational tie resulted from having a choice of whether to first devoice the first stop or the second stop and lacking a constraint that distinguishes between this choice. Thus, one derivational path first devoices the first stop /b/ to [p] and then devoices the second stop [d] to [t] at the next step in the derivation. The other derivational path — in the reverse order — first devoices /d/ to [t] and then [b] to [p]. Both paths converge on optima that are identical [pata]. The two possible paths in full for input /bada/ are below:

a) /bada/ → [**p**ada] → [pa**t**a] → [pata]
b) /bada/ → [ba**t**a] → [**p**ata] → [pata]

OT-Help currently implements the *multiple input* method as explained in section 1.2, which places the outputs from tied derivational paths into a single tableau. This means that formally there can be more than one output winner for a given derivation.

In typology windows, multiple winning outputs derived from the multiple derivational paths are displayed in a single output cell separated by commas. In this example (Fig. 13), there are two such outputs, which happen to be identical:

pata, pata

In short derivation windows, both of the tied derivational paths are displayed as shown in (Fig. 14).

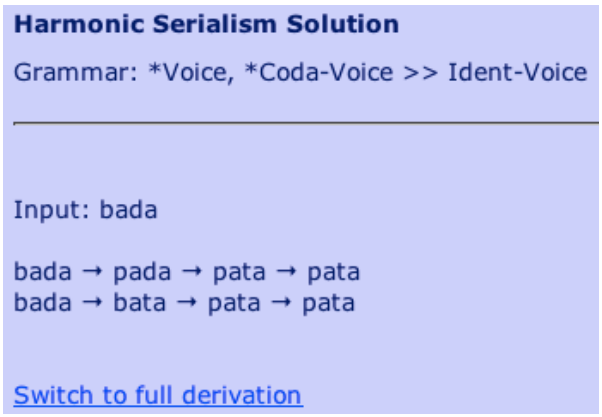

Harmonic Serialism Solution
Grammar: *Voice, *Coda-Voice >> Ident-Voice
Input: bada
bada → pada → pata → pata
bada → bata → pata → pata
Switch to full derivation

Fig. 14: Serial OT short derivation window for language with tied multiple outputs.

In the full derivation window for this input (Fig. 15), two output candidates are tied as the winners at the first step of the derivation. Due to OT-Help's multiple input implementation, both winners become inputs to the second step in the same tableau, and output candidates are derived from them. The winners of the second step are then inputed into third tableau. Since the winners of the second step are identical, OT-Help generates the candidate set from the winners only once in the third step. The fourth step of the derivation converges on a single output although this output was derived from two tied inputs at earlier steps.

A similar situation holds if there are multiple optima that are not identical.

2.10.2 Input Death

There is another possible outcome of a derivational tie. In a subsequent step of the derivation with the tie, an output from one of the tied outputs at the previous step can be the optimum while all outputs from another tied output at the previous step are losing candidates. This is a possibility because OT-Help currently implements the *multiple*

Input: bada

Grammar: *Voice, *Coda-Voice >> Ident-Voice

Step 1:

Input: bada	*Voice	*Coda-Voice	Ident-Voice
bada	-2	0	0
☞ pada	-1	0	-1
☞ bata	-1	0	-1

Step 2:

Input: pada, bata	*Voice	*Coda-Voice	Ident-Voice
pada → pada	-1	0	-1
pada → bada	-2	0	-2
☞ pada → pata	0	0	-2
bata → bata	-1	0	-1
☞ bata → pata	0	0	-2
bata → bada	-2	0	-2

Step 3:

Input: pata, pata	*Voice	*Coda-Voice	Ident-Voice
☞ pata → pata	0	0	-2
pata → bata	-1	0	-3
pata → pada	-1	0	-3

Step 4:

Input: pata	*Voice	*Coda-Voice	Ident-Voice
☞ pata	0	0	-2
bata	-1	0	-3
pada	-1	0	-3

Fig. 15: Serial OT full derivation window for language with tied multiple outputs.

input method instead of forking each tied outputs into separate tableaux (i.e., the *branching derivation* method) as shown in section 1.2. When a tied output derives only losing

outputs at a later step, this situation is referred to as *input death* (section 1.2.1). Probably in most cases, input death signals the user that their constraint set is inadequate.

Using the example of section 1.2.1 to demonstrate show OT-Help displays input death, a vowel stressing operation is added to the set of operations. We represent a stressed vowel as a capital `A` and an unstressed vowel as a lowercase `a`. And, the constraint set includes our previous `Ident-Voice` faithfulness constraint (Fig. 4) and the new constraints `OCP-Voice`, `Stress-Right`, and `*Stress-Voi-Ons`. `OCP-Voice` assigns a violation to words with multiple voiced obstruents. `Stress-Right` assigns a violation to final vowels that are unstressed. `*Stress-Voi-Ons` assigns a violation to stressed vowels that have a voiced onset. The user files for this example are the following:[10]

```
badaTieDeath.txt
badaTieDeath.txt_OPERATIONS
badaTieDeath.txt_CONSTRAINTS
```

The serial OT typology window for this example is shown in Fig. 16.

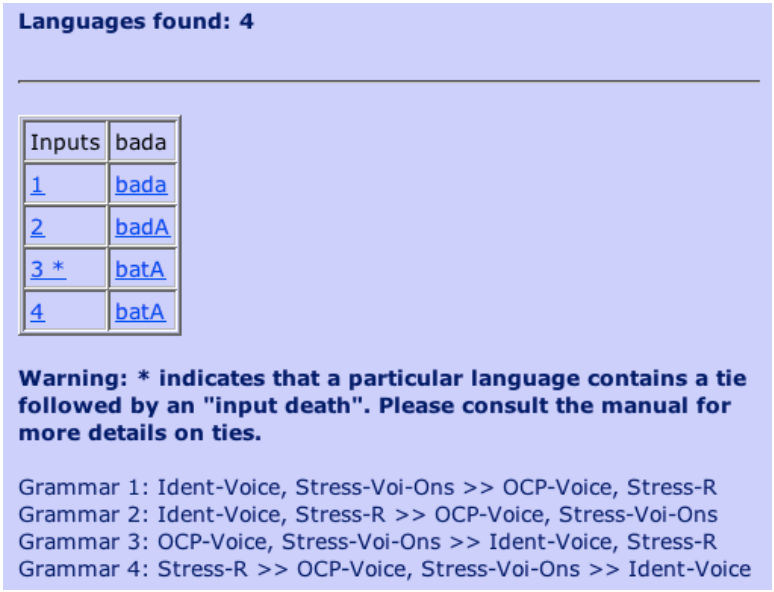

Languages found: 4
Inputs
bada
1
bada
2
badA
3 *
batA
4
batA
Warning: * indicates that a particular language contains a tie followed by an "input death". Please consult the manual for more details on ties.
Grammar 1: Ident-Voice, Stress-Voi-Ons >> OCP-Voice, Stress-R
Grammar 2: Ident-Voice, Stress-R >> OCP-Voice, Stress-Voi-Ons
Grammar 3: OCP-Voice, Stress-Voi-Ons >> Ident-Voice, Stress-R
Grammar 4: Stress-R >> OCP-Voice, Stress-Voi-Ons >> Ident-Voice

Fig. 16: Serial OT typology window with input death language.

[10] We do not show how these new operation and constraints are implemented in regular expressions in order to keep this discussion focused on window display.

At this point, OT-Help gives a warning message in bold between the language table and the list of constraint rankings/weightings. Additionally, the language(s) that suffer from input death are indicated in the language table with an asterisk (*) after their language number. In Fig. 16, the typology window shows that Language 3 with its apparently single output `batA` has another input derived from a tied output at an earlier step that suffered from input death and as a result produced only losing outputs. This input that died can be seen in the typology and derivation windows.

In the short derivation window for Language 3 (Fig. 17), the single output optima is indicated with a following asterisk. The derivational path that died is indicated with a dagger (†). OT-Help also displays warning messages indicating this information.

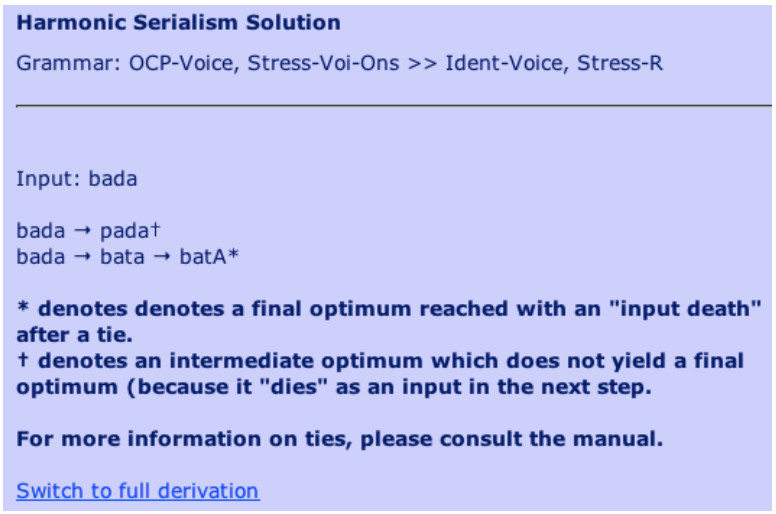

Harmonic Serialism Solution
Grammar: OCP-Voice, Stress-Voi-Ons >> Ident-Voice, Stress-R
Input: bada
bada → pada†
bada → bata → batA*
* denotes denotes a final optimum reached with an "input death" after a tie.
† denotes an intermediate optimum which does not yield a final optimum (because it "dies" as an input in the next step.
For more information on ties, please consult the manual.
Switch to full derivation

Fig. 17: Serial OT short derivation window for language with input death.

In the full derivation window for Language 3 (Fig. 18), in addition to the optimum, all losing output candidates that are derived from the *dead* input are indicated with a double dagger (‡). At the bottom of the tableaux set is a warning message.

2.11 Printing

There is currently no method for printing from within OT-Help.

Input: bada

Grammar: OCP-Voice, Stress-Voi-Ons >> Ident-Voice, Stress-Right

Step 1:

Input: bada	OCP-Voice	Stress-Voi-Ons	Ident-Voice	Stress-Right
bada	-1	0	0	-1
☞ pada	0	0	-1	-1
☞ bata	0	0	-1	-1
badA	-1	-1	0	0
bAda	-1	-1	0	-1

Step 2:

Input: pada, bata	OCP-Voice	Stress-Voi-Ons	Ident-Voice	Stress-Right
‡pada → pada	0	0	-1	-1
‡pada → bada	-1	0	-2	-1
‡pada → pata	0	0	-2	-1
‡pada → padA	0	-1	-1	0
‡pada → pAda	0	0	-1	-1
bata → bata	0	0	-1	-1
bata → pata	0	0	-2	-1
bata → bada	-1	0	-2	-1
☞ bata → batA	0	0	-1	0
bata → bAta	0	-1	-1	-1

Step 3:

Input: batA	OCP-Voice	Stress-Voi-Ons	Ident-Voice	Stress-Right
☞ batA	0	0	-1	0
patA	0	0	-2	0
badA	-1	-1	-2	0

‡ denotes a candidate deriving from an input that contributes no optimum in the step.

For more information on ties, please consult the manual.

Fig. 18: Serial OT full derivation window for language with input death.

Helpful Hint

There is a roundabout method of printing OT-Help windows. On Windows, pressing the keyboard shortcut Alt-PrtSc saves the image of any OT-Help window to the clipboard. The image may then be pasted into a word processor file.

On Macs, the keyboard short Command-Shift-4 allows you to draw a marquee to save the selection as a file on the desktop. Command-Control-Shift-4 followed by space saves the image to the clipboard.

In both systems, you must make sure your OT-Help window is resized to the size you want as the image saved will be exactly as it appears on your monitor.

2.12 Saving and Exporting

There is currently no direct method for saving or exporting information in OT-Help to an external file.[11]

Helpful Hint

Although there is no direct method for saving OT-Help data, tableaux and other text in OT-Help windows can be selected and copy-&-pasted into spreadsheet programs, text editors, and word processors.

3 Operations

Operations generate the candidate set and specify the violations of faithfulness constraints.

Operations are defined in the operations file.

[11] Exporting features are currently planned for future releases.

All of the operations seen in this section can be found within the `Section3Op` directory in the `.ZIP` file at http://web.linguist.umass.edu/~OTHelp/OTHexs.zip. In order to run the operation files, associated input and constraint files are required, which we have also provided. The input strings are the same as seen in this section. The constraint files only contain the faithfulness constraints that are associated with the particular operation. The output candidates generated by the operations appear in OT-Help under the *full derivation* window.

3.1 Operation Filename & Extension

The filename of the operation file must be the same as the filename of the input file and the constraint file.

Additionally, the operation file must the extension `.txt_OPERATIONS`.

Neither the filename or the file extension is case-sensitive.

3.2 Operation File Format

The operations file is a tab-delimited plain text file. The three columns in the operations file are separated by a *single tab*.

The file consists of a list of operations followed by an end of file marker `[end operations]`.

Each operation begins with the operation label `[operation]` and is followed by a set of parameters with arguments.

It is not necessary to skip a line (carriage return) between operations (although doing so increases human readability).

3.3 Operation Parameters

Each operation consists of five parameters:

```
[operation]
[long name]        < operation name >
[active]           < yes | no >
[definition]       < transformation >
[violated faith]   < constraint name >
```

Each parameter must appear on a separate line. The `[operation]` parameter has no argument, but the other parameters *do* have arguments. Each parameter and its argument are separated by a tab.

The first four parameters — `[operation]`, `[long name]`, `[active]`, `[definition]` — are obligatory. The `[violated faith]` parameter is optional.

Operation Label `[operation]`. The operation label marks the beginning of an operation and doesn't have any arguments. The operation label is followed by a carriage return. Note that extraneous whitespace after the label will cause constraints to become unreadable by OT-Help.

Operation Name `[long name]`. This argument is obligatory. Currently, an operation's name does not show up anywhere within OT-Help. If this parameter is missing, the operation is ignored by OT-Help.

Activation Switch `[active]`. The activation switch determines whether an operation is active in serial derivations. It has a value of `yes` or `no`. A `yes` setting causes the operation to be used in serial derivations. A `no` setting causes the operation to be ignored. This setting can be used to test the effects of different sets of operations without removing any operations from the file. If this parameter is missing, the operation is ignored by OT-Help.

Operation Definition `[definition]`. Operation definitions are formally transformations similar to the transformations used in early generative grammar. The definition

parameter consists of a structural description argument and a structural change argument. The structural description argument contains a one or more indexed regular expressions. Any substrings that match the structural description are rewritten by the indexed strings specified in the structural change argument.

[definition] < *structural description* > < *structural change* >

The [definition] parameter name, the structural description argument, and the structural change argument must separated from each other by a single tab. An operation can have more than one [definition] parameter.

If the [definition] parameter is missing, the operation is ignored by OT-Help.

Operation to Faithfulness Constraint Link [violated faith]. This parameter specifies which faithfulness constraint is violated by performing the operation. The argument of this parameter must be the constraint's [long name], which is case-sensitive. The faithfulness constraint also has to be present and active in the constraint file. The name of the faithfulness constraint must be exactly the same in both files.

This parameter is optional.

An operation can violate multiple faithfulness constraints by adding multiple [violated faith] parameters on different lines.

3.4 Context-free Operations

This section provides several examples of operations in order to demonstrate how to write the transformations used in operation definitions.

Helpful Hint

Since [`long name`] arguments in constraints are case-sensitive in OT-Help, you must take care that the name occurring in the the operation's [`violated faith`] parameter and the constraint's [`long name`] parameter is exactly the same.

For example, if you have [`violated faith`] with an argument of `IdentNasal` in the operation file but [`long name`] with an argument of `identnasal` in the constraint file, the constraint and operation would not be associated with each other because `IdentNasal` and `identnasal` are different names. In this situation, the `identnasal` constraint would appear in typologies but not assign violations to any candidates, and `IdentNasal` would not appear anywhere.

Helpful Hint

A general point about making operation and constraints for your typologies in OT-Help: It is typically much easier to use schematic examples using only ASCII characters. Firstly, this will reduce the number of symbols you need to deal with. Secondly, writing regexps often becomes easier, and, practically always, they are easier to read afterwards.

For instance, if a constraint against stress clash is desired, you probably only need one character for a stressed syllable (e.g. `x`) and a second character for an unstressed syllable (e.g., `o`). With this limited repertoire, your regexps only need to involve two symbols. The *Clash regexp could be as simple as matching two-character sequences of `xx` (assuming a grid representation without metrical feet). To give another example, if you are modeling a language with several vowels (like English or Vietnamese), it is not necessary to include every vowel in your OT-Help representation: only a few characters may suffice. To think about it another way, since languages operate on natural classes of sounds, you can use a single character per natural class.

However, in order to introduce new users to regexps (hopefully gently), we have not always pared representations to the absolute schematic minimum in this user guide.

3.4.1 Character Replacement

In the first example below, lowercase vowels, representing oral vowels, are replaced by their uppercase counterparts, representing nasal vowels. A candidate generated by `Nasalization` violates both `Ident` and `IdentNasal`, both of which would be identified in the constraint file.

```
[operation]
[long name]       Nasalization
[active]          yes
[definition]      a            A
[definition]      i            I
[definition]      u            U
[violated faith]  Ident
[violated faith]  IdentNasal
```

Some examples of the output of the `Nasalization` operation are below.

Input	Candidate(s) Generated by `Nasalization`
`ib`	`Ib`
`putu`	`pUtu` , `putU`
`abubi`	`Abubi` , `abUbi` , `abubI`

3.4.2 Character Deletion

In the second example below, lowercase vowels are rewritten with an empty string (i.e., they are deleted). The structural description of the argument is the regexp `[aeoiu]` while the structural change argument is blank. A candidate generated by `VowelDeletion` violates both `Max` and `MaxVowel`, both of which would be identified in the constraint file.

```
[operation]
[long name]        VowelDeletion
[active]           yes
[definition]       [aeoiu]
[violated faith]   Max
[violated faith]   MaxVowel
```

Helpful Hint

In a definition with no structural change argument, a tab is not required after the structural description argument.

Some examples of the output of the `VowelDeletion` operation are below.

Input	Candidate(s) Generated
eb	b
poto	pto, pot
abubo	bubo, abbo, abub

Helpful Hint

A useful regexp idiom is the *character class*. We used a character class in the `VowelDeletion` operation. A character class is simply a set of characters — any of which may match a character in the search string. To create one, you list all the characters in the character class (without spaces) and enclose them in square brackets. The character class `[aeoiu]` will match every a, e, o, i, or u occurring in the string.

Since `Max` is violated by all instances of deletion, it is also violated by the operation `DeleteC`.

```
[operation]
[long name]        DeleteC
[active]           yes
[definition]       [ptkbdgmnrlwj]
[violated faith]   Max
```

Some examples of the output of the `DeleteC` operation are below.

Input	Candidate(s) Generated
eb	e
poto	oto , poo
abubo	aubo , abuo

HELPFUL HINT

It is probably never a good idea to define two operations that overlap in their effects. For example, if you were using something similar to our VowelDeletion and DeleteC operations, then there is no need for a general deletion operation that deletes both vowels and consonants.

Of course, you may very well want a general MAX constraint, but this general faithfulness constraint can simply be associated with both the VowelDeletion and DeleteC deletion operations as we have done here.

3.4.3 Character Insertion

Simple insertion operations insert characters at the beginning and end of the candidate string and at every position between the characters that occur in the string. For the definition, the structural description argument is blank, and the structural change argument contains the insertion character(s). The Epenthesis operation below demonstrates this: candidates are generated by inserting the vowel a at every position in the word.

```
[operation]
[long name]       Epenthesis
[active]          yes
[definition]                  a
[violated faith]  Dep
```

Some examples of the output of the Epenthesis operation are below.

Input	Candidates Generated
eb	aeb , eab , eba
buk	abuk , bauk , buak , buka
poto	apoto , paoto , poato , potao , potoa

An operation with a blank structural description still requires a tab between the (blank) structural description argument and the structural change argument. OT-Help assumes that a definition with only one argument has a structural description argument and no structural change argument. The tabless definition below will produce a deletion operation instead of an insertion operation.

Tabbed definition (insertion): `[definition]` (tab) `abcd`

Tabless definition (deletion): `[definition]` `abcd` (no tab)

HELPFUL HINT

When creating character inserting operation, you must include a faithfulness constraint that is violated by the insertion operation. If you do not, then OT-Help will get stuck in an infinite loop, and you must force the program to quit. Since OT-Help currently provides no error message feedback to the user, OT-Help will appear to do nothing when in an endless loop.

3.5 Context Dependency & Substring Indexing

The structural change argument can reference and copy the substring(s) matched by a regexp in the structural description argument. When an operation applies, the substring(s) matching the regexp(es) in the structural description argument are stored and assigned indices. By using the indices in the structural change argument of the operation, the stored strings can be used to rewrite the substrings matched by the structural description argument.

The structural description argument of `[definition]` determines how indices are assigned. Indices are assigned left-to-right starting with 1 to each regular expression separated by an underscore (_).

The operation definition below assigns the following index to the substring match:

```
[definition]  abc  < structural change >
```

```
1
abc
```

The operation definition below assigns the following indices to the substring match:

```
[definition]  a_bc  < structural change >
```

```
1  2
a  bc
```

The operation definition below assigns the following indices to the substring match:

```
[definition]  a_b_c  < structural change >
```

```
1  2  3
a  b  c
```

Each indexed regular expression can be represented in the structural change argument by a number corresponding to its index. A *single space* separates each index number from other material in the structural change argument.

Helpful Hint

The structural change argument must not contain more indices than are defined in the structural description argument.

The structural change argument can contain both indices and other characters. This allows for context-dependent operations. In the operation below, an a is inserted between stops.

```
[operation]
[long name]       CaCInsertion
[active]          yes
[definition]      [pbtdkg]_[pbtdkg]  1 a 2
[violated faith]  DepCaC
```

Some examples of the output of the `CaCInsertion` operation are below.

Input	Generated Candidate(s)
bd	bad
ptd	patd, ptad
bbtog	babtog, bbatog

Indices in the structural change argument can be re-ordered to create operations that switch the order of segments. In the operation below, a stop-vowel sequence is metathesized.

```
[operation]
[long name]       Metathesis
[active]          yes
[definition]      [pbtdkg]_[aeoiu]  2 1
[violated faith]  Linearity
```

Some examples of the output of the `Metathesis` operation are below.

Input	Generated Candidate(s)
be	eb
poto	opto, poot
abubo	aubbo, abuob

Indices can be omitted from the structural change argument to create operations that delete segments in a specific context. If indices are missing in the structural change argument, the operation won't write any missing indexed substring(s) when it rewrites the input string. In the operation below, intervocalic stops are deleted.

```
[operation]
[long name]       IntervocalicStopDeletion
[active]          yes
[definition]      [aeoiu]_[pbtdkg]_[aeoiu]  1 3
[violated faith]  MaxIntervocalicStop
```

Note: In a definition with missing indexed structural change numbers, no extra spaces are required for the missing numbers. `1 3` is okay, with one space, and so is `1  3`, with two spaces.

Some examples of the output of the `IntervocalicStopDeletion` operation are below.

Input	Generated Candidate(s)
ebo	eo
poto	poo
abubo	aubo , abuo

4 Constraints

Constraints assign violation penalties (or rewards) to output candidates.

Most constraints are defined in the constraint file. However, `faithfulness` constraints are defined in the operation file, and *pre-defined* constraints are not defined at all but are rather called by name in the input file.

In addition to defining, all constraints (except for pre-defined constraints) must be turned on (be made *active*) from within the constraint file.

OT-Help allows six types of constraints:

1) markedness
2) alignment
3) scalar
4) long distance markedness
5) faithfulness
6) pre-defined

In this section, we explain and exemplify how constraints work in OT-Help. First, we provide some general notes that apply to user-defined constraints. The parameters for user-defined constraints are explained next (section 4.3.1). Then, we show examples of each constraint type (section 4.4).

All of the constraints seen in this section can be found within the `Section4Con` directory in the `.ZIP` file at http://web.linguist.umass.edu/~OTHelp/OTHexs.zip. In order to run the constraint files, associated input and operation files are required, which we have also provided. The input strings are the same as the candidate strings seen in this section. (Each candidate string will show up in OT-Help as separate input. We have done this for these examples in order to just show the assigned violations without the complications of derivations and candidate competition.) The operation files do not contain any operations. The violations assigned by the particular constraint appear in OT-Help under the *full derivation* window.

4.1 Constraint Filename & Extension

The filename of the constraint file must be identical to the filename of the input file and the operation file.

Additionally, the constraint file must have the extension `.txt_CONSTRAINTS`.

Neither the filename or the file extension is case-sensitive.

4.2 Constraint File Format

The constraint file is a tab-delimited text file consisting of a list of constraints followed by an end of file marker [`end constraints`].

We used `bakad.txt_CONSTRAINTS` (Fig. 4) as our example constraint file in section 2.

It is not necessary to skip a line (carriage return) between constraints (although doing so increases human readability).

4.3 Format of User-defined Constraints

A user-defined OT-Help constraint consists of a sequence of parameters.[12] Each parameter consists of a parameter name and a parameter argument. When parameters are used, the parameter name is obligatory and most (but not all) parameters require arguments as well.

Only a *single tab* can occur between a parameter name and its argument.[13]

A user-defined constraint can have six parameters: (1) the beginning of constraint marker [`constraint`], (2) the constraint name [`long name`], (3) an optional abbreviated name for OT-Help window display [`short name`], (4) the activation switch [`active`], (5) the constraint type [`type`], (6) the constraint definition [`definition`], and (7) numerical violation assignment adjuster [`violation pattern`].

Only the [`constraint`], [`long name`], and [`type`] parameters are obligatory for all user-defined constraints.

The generalized constraint structure is shown below.

[12] Pre-defined constraints, in contrast, are only called by name, and, thus, their definitions are hidden from the user.

[13] There is no error message informing the user of tabbing errors. This type of error can cause the constraint to become unreadable (and consequently absent from typologies).

[constraint]	
[long name]	< *name* >
[short name]	< *name or default to* long name >
[active]	< yes | no >
[type]	< markedness | alignment | scalar | long-distance | faithfulness >
[definition]	< *regexp(s) or alignment statement* >
[violation pattern]	< *list of violation weight(s) or default pattern* >

A real example of the *Coda-Voice markedness constraint used in section 2 for our simple typology example of section 1 is repeated below.

```
[constraint]
[long name]   *Coda-Voice
[active]      yes
[type]        markedness
[definition]  [bdg]$
```

4.3.1 Parameters

Each parameter consists of a parameter name and one or more parameter arguments. A single tab occurs between the parameter name and its argument.

The [constraint], [long name], and [type] parameters are obligatory for all user-defined constraint types.

The [definition] parameter is obligatory for all user-defined constraint types except for faithfulness constraints.

Parameter names are not case-sensitive. Therefore, [constraint], [Constraint], and [CONSTRAINT] are all equivalent instances of the [constraint] parameter name.

Constraint Label [constraint]. This parameter is a *start of constraint* marker used to separate constraints in the constraint list. It does not take an argument, so any whitespace (including a tab) or alphanumeric character following the [constraint] marker is ignored. This parameter is obligatory.

Constraint Name [long name]. This parameter is used to name constraints. Its argument is a string of alphanumeric characters. (Any character is ok.)

The argument is case-sensitive, so NoCoda, nocoda, and Nocoda are all different constraint names.

This parameter is obligatory. If the [long name] parameter is completely absent or if its argument is absent, then the constraint becomes unreadable and is ignored in typologies.

The constraint's [long name] will appear in OT-Help typology and derivation windows by default if no [short name] parameter is included. However, if [short name] parameter is defined, then the [long name] will not be displayed in OT-Help windows.[14]

Helpful Hint

Care must be taken when naming constraints so that they always have unique names. If two constraints have the same name, then the second constraint with override the first constraint: Only one constraint with this name will appear in typology calculations, and it will assign violations according the second constraint in the constraint file.

Connecting Operations to Faithfulness Constraints. In the operation file, operations can be tied to faithfulness constraints via a [violated faith] operation parameter. This [violated faith] parameter refers to the [long name] (and not the [short name]) of the faithfulness constraint inside in the constraint file.

Display Name [short name]. This parameter provides the constraint name that appears in OT-Help typology and derivation windows.

[14] Users could exploit this behavior to use the [long name] as a description of what the constraint does and the [short name] as the displayed name if desired.

Helpful Hint

Because the [long name] is case-sensitive, care should be exercised so that the [violated faith] and [long name] arguments exactly match in both the operation and constraint files. If the [violated faith] and [long name] argument do not match, then this particular faithfulness constraint will still appear in typologies, but it will not assign violations of performing the intended operation to candidates. In this case, it will probably result in the faithfulness constraint not assigning violations at all (although a copy-&-paste error could result in this faithfulness constraint assigning violations of a different unintended operation.)

The [short name] parameter is optional. If it is absent, then the [short name] will default to the value of the [long name] parameter.

Activation Switch [active]. The active switch can have an argument value of either yes or no. A yes setting activates the constraint for use in serial typologies while the no setting allows the constraint to be ignored, and it will not appear in OT-Help typology or derivation windows. The [active] parameter may be useful when trying out different constraint definitions or when examining constraint interaction.

The [active] parameter is optional. If it is absent, then OT-Help will default to the [active] = yes setting.

The [active] argument (yes/no) is not case-sensitive. However, if the argument is misspelled, then the constraint becomes unreadable and is ignored in typologies.

Constraint Type [type]. The [type] parameter has five possible values: (1) markedness, (2) alignment, (3) scalar, (4) long-distance, and (5) faithfulness.

This parameter is obligatory. If it or its argument is absent, then the constraint becomes unreadable and is ignored in typologies.

The `markedness`, `alignment`, `scalar`, and `long-distance` constraint types require a `[definition]` parameter which must follow the `[type]` parameter.

Information about each of the constraint types is detailed in section 4.4.

Constraint Definition `[definition]`. This parameter is used to define which substring matches are considered violations. Its format is dependent upon the constraint type. `Markedness` constraints have a definition which is a positive lookahead regular expression. `Alignment` constraint definitions have the form of generalized alignment constraints. `Scalar` constraints have an ordered sequence of regular expressions of which each member of the sequence potentially receives a different numerical violation value.

Each of these four constraint types (`markedness`, `alignment`, `scalar`, `long-distance`) require the `[definition]` parameter. If it is absent, then the constraint becomes unreadable and is ignored in typologies.

The `faithfulness` constraint type does not require the `[definition]` parameter. Including it will have no effect on this constraint type.

Numerical Violation Adjuster `[violation pattern]`. By default, constraints assign a penalty value of −1 for each violation mark as is standard OT practice. However, the default behavior can be modified via the `[violation pattern]` parameter. The numerical argument of this parameter is multiplied by the number of violation marks. For example, if the value of `[violation pattern]` is set to `5`, then a candidate that receives 4 violations by default will receive a penalty of −20 (5 x −4 = −20), which will be displayed in OT-Help tableaux.

Additionally, the `[violation pattern]` may be negative. A negative value will result in a positive constraint (i.e., a constraint that rewards instead of penalizes). For example, if the value of `[violation pattern]` is set to `-5`, then a candidate that receives 4 violations by default will receive a reward of +20 (−5 x −4 = 20), which will be displayed in OT-Help tableaux.

This parameter is optional.

Constraints of type `scalar` differ slightly. By default, the penalities assigned increase incrementally + −1 for each step in the scale (e.g., −1, −2, −3, …). Using the `[violation pattern]` parameter allows for any arbitrary numerical value to be assigned as the violation incurred by a given step of the scale. As with other constraint types, these may be turned into positive rewards. If the optional `[violation pattern]` parameter is absent, the `scalar` constraint follows the default pattern.

4.4 Constraint Types

Here we show examples of each type of markedness constraint detailing the required parameters.

4.4.1 `Markedness`

This type of constraint is defined as a regular expression. Every substring that matches the regular expression (regexp) will incur a violation. In other words, the number of matches found by the regexp search through the output candidate string equals the number of violations that is assigned to that output string candidate.

The sense of the OT-Help term *markedness* is not strictly the same as its sense in the OT literature. Rather, here it means a non-alignment, non-scalar markedness constraint that can be defined solely via a regexp matching search.

Here the `[constraint]`, `[long name]`, and `[definition]` parameters are obligatory. And, the `[type]` parameter must be set to `markedness`.

Examples

For a trivial example, a constraint `[definition]` of `x` will find three matches in an output string `xxx` and three violations will accordingly be assigned to the output candidate `xxx`.

```
[constraint]
[long name]   Lowercase x Is Bad
[short name]  *x
[active]      yes
[type]        markedness
[definition]  x
```

Since we want the constraint to be used in typologies, the [active] parameter argument is set to yes. Since the long name is a bit long, we want an abbreviated name to occur in OT-Help windows, so the [short name] parameter is set to *x: now this constraint will only be shown with the name *x in OT-Help. Since this is a markedness constraint, its [type] is set to that value. The matches found with this simple [definition] regexp is below:

Output Candidate	Number of Matches	Substring Matches
xxx	3	x, x, x

A more complex example of *NC̥ follows. This *NC constraint assigns a violation for every sequence of a nasal followed by a voiceless obstruent.

```
[constraint]
[long name]   *NC
[active]      yes
[type]        markedness
[definition]  [mnN][ptk]
```

Our regexp [mnN][ptk] will find the following matches given the following candidate strings.

Candidate	Number of Matches	**Substring Match(es)**
lamti	1	mt
unpe	1	np
roNkampir	2	Nk , mp
dukkeg	0	
onna	0	
ikmul	0	
wenod	0	
api	0	

We will provide one further and more complex constraint example: Share. The Share constraint assigns a violation to every pair of segments that are not linked to the same instance of a particular distinctive feature. In our flat representation, we indicate segments that are linked to this feature as capital alpha characters and segments that are *not* linked to this feature as lowercase alpha characters.[15] Thus, two-character sequences of capital + lowercase (e.g., Ab), lowercase + capital (e.g., aB), and lowercase + lowercase (e.g., ab) will incur a violation mark. But, a sequence of two capital letters (e.g., AB) will *not* receive a violation.

This constraint, then, is the following:

```
[constraint]
[long name]   Share
[active]      yes
[type]        markedness
[definition]  [A-Z][a-z]|[a-z][A-Z]|[a-z][a-z]
```

And, given the candidate strings below, the following matches are found:

[15] We are not concerned with the precise feature here although the feature was [nasal] in McCarthy (2009, to appear). For simplicity, we assume here that all segments linked to this feature (i.e., capital letters) are instances of the same feature.

Helpful Hint

Two regexp idioms are introduced in the definition of `Share`. First is the capital/lowercase shorthand. The set of all capital letters (in the Roman alphabet) is written as the character class `[A-Z]` while the lowercase character class is `[a-z]`. So, to match any sequence of a capital letter followed by a lowercase letter, you can use the regexp `[A-Z][a-z]`.

The second idiom is the *metacharacter* for the alternation operator, which is the vertical bar symbol `|`. If you want to match either regexp *A* or regexp *B* or regexp *C* and so on in a single regexp, then you place the alternation operator between the individual regexps. For instance, if you want to match `pa` or `ta` or `ka`, then the regexp with the alternation operator is `pa|ta|ka`. In the `Share` definition, we wanted to match the sequence `[A-Z][a-z]` or the sequence `[a-z][A-Z]` or the sequence `[a-z][a-z]`. Hence, our use of the alternation operator.

Candidate	Number of Matches	**Substring Match(es)**
`bayi`	3	`ba , ay , yi`
`Mayi`	3	`Ma , ay , yi`
`MAyi`	2	`Ay , yi`
`MAYi`	1	`Yi`
`MAYI`	0	
`pukEN`	3	`pu , uk , kE`

4.4.2 `Alignment`

`Alignment` constraints are not defined in terms of regexp matching searches, but, instead, follow the format of generalized alignment (McCarthy & Prince 1993). The generalized alignment statement may include arguments that themselves contain regexps.

The `[constraint]`, `[long name]`, and `[definition]` parameters are obligatory. And, the `[type]` parameter must be set to `alignment`.

Technical Note

`Markedness` Constraint Regexps are Positive Lookaheads

Specifically, the argument of the `[definition]` parameter is a *positive lookahead* (or *positive forward zero-width assertion*) that is suffixed to an empty string. Wrapping the argument in a lookahead allows for the constraint to match substrings in the standard way used in the OT literature.

As a simple example, a constraint `*VV` assigns a violation for every two-vowel sequence. Given the candidate string `aiu`, we want this `*VV` constraint to find two matches as shown below.

Candidate	Number of Matches	Substring Matches
`aiu`	2	`ai , iu`

This constraint's apparent regexp definition is as follows:

`[definition]` `[aiu]{2}`

This will match pairs of characters consisting of any combination of characters within the character class. As the argument is in a positive lookahead, the regexp is actually the following:

`(?=[aiu]{2})`

(If the regexp argument was not enclosed in a lookahead — that is, if the regexp was merely `[aiu]{2}` — then only one match (the first match `ai`) would be found. This is not the desired matching behavior.)

OT-Help automatically wraps regexp arguments in a positive lookahead. The user should just keep in mind that any `markedness` constraint's `[definition]` apparently in the form `xyz` is technically a regexp of the form `(?=xyz)`. (This also means that the regexp is not actually matching the substrings `ai` and `iu` in this example, but rather the zero-width position anchors that precede these lookahead substrings, which are consumed as the regexp search proceeds.)

This particular behavior only applies to `markedness` type constraints. All other contraints in OT-Help do not automatically wrap regexps into positive lookaheads. (Obviously, this is also true of operations.)

The `[definition]` format of the `alignment` constraint type is the following. The definition has four arguments:

substring aligned alignment edge direction substring counted

Each argument is separated by a *single tab*. The first argument is the item that is being aligned. The second argument is the alignment edge: this is the goal or endpoint of the alignment and is typically the edge of a phonological domain. In other words, the first argument item wants to be perfectly aligned with the second argument alignment edge. The direction argument indicates alignment direction — either `right` or `left`. `Alignment` constraints assign violations according to the sum of items that intervene between each of the aligned items and the alignment edge — the final argument identifies these intervening items used for counting the violations.

Examples

As an example, we use the standard ALL-FEET-RIGHT alignment constraint.[16] This constraint assigns a violation for every syllable that intervenes between the right edge of every foot and the right edge of the prosodic word ("Every foot stands in final position in the PrWd").

For our flat representation, we indicate feet as parentheses — an open parens < `(` > for the left edge, a closed parens < `)` > for the right edge. The syllables are either `x`s or `o`s. And, the prosodic word boundaries will not be explicitly marked, but we can refer to these edges as the beginnings and ends of our output candidate strings. As an example candidate, the string `(xo)(xo)o` has two left-aligned trochaic feet with nonexahustive parsing. According to ALL-FEET-RIGHT, it receives a total of four violation marks.

In terms of the `alignment[definition]` format, the items being aligned are feet, particularly the right edge of the feet, so the aligned substring is a closed parens, which is escaped as `\)`. The alignment edge is the end of the candidate string, which can be indicated with the end of line regexp metacharacter `$`. The third direction argument is `right`. The items intervening between `\)` and `$` belong to the character class `[xo]`, which is the fourth argument.

[16] This has also been called ALIGN-RIGHT(foot, word) or ALIGN(foot, right; PrWd, right).

```
[constraint]
[long name]   AllFeetRight
[short name]  AllFtR
[active]      yes
[type]        alignment
[definition]  \)           $  right  [xo]
```

In other words, this alignment statement counts all xs and os that occur between each) and the end of the string and then adds all of the counts up to return the total violation value. Given our candidate string of (xo)(xo)o, the first (leftmost) closed parens has three counting items that intervene between it and the end of the string, which are indicated by subscript numbers:

$$(xo)(x_1o_2)o_3$$

The second (rightmost) closed parens has one counting item that intervenes between the parens and the end of the string:

$$(xo)(xo)o_1$$

The total number of violations assigned by `AllFeetRight` is then four ($3 + 1 = 4$).

Aligning from the opposite direction gives us `AllFeetLeft`:

```
[constraint]
[long name]   AllFeetLeft
[short name]  AllFtL
[active]      yes
[type]        alignment
[definition]  \(           ^  left  [xo]
```

The counting substrings (the fourth argument) are the same. The aligned substring is now the (escaped) open parens `\(`. Our alignment edge is the beginning of line regexp

Helpful Hint

Regexps have reserved symbols — known as *metacharacters* — which have special functions. For example, the beginning and end of lines are referenced via the metacharacters ^ and $, respectively.

Unfortunately, some metacharacters are typically used with special meanings in phonological representations. Because of this, many users may want to treat these characters not as metacharacters but as regular characters. In order to do this, these metacharacters must be *escaped*. *Escaping* simply means that the escape symbol — the backslash \ — must precede the metacharacter you wish to treat as a normal character. So, $ is the end of a line metacharacter but \$ is just a dollar sign.

In phonological transcription, periods often represent syllable boundaries and parentheses, square brackets, and curly braces often represent various prosodic or morphological boundaries. And, they all happen to be regexp metacharacters. If you use them in your input strings, these all need to be escaped:

```
\.
\(      \)
\[      \]
\{      \}
```

metacharacter ^, and the direction is `left`. Given the same candidate string (xo)(xo)o, this alignment statement counts two counting items: (x_1o_2)(xo)o.

A more complicated stress alignment constraint is `AlignRightHeadFoot`, which aligns the foot that heads the prosodic word (i.e., the foot carrying main stress) to the right edge of the prosodic word domain. The syllable with main stress can be indicated with a capital X (as opposed to secondary stress with a lowercase x). The head foot is, then, the foot that contains X. Our alignment statement is largely the same as `AllFeetRight`, but our aligned item argument has changed to \(o?Xo?\).[17]

[17] We are making the simplifying assumption that feet are maximally disyllabic. We would need to modify our regexp in order to match larger polysyllabic head feet.

```
[constraint]
[long name]   AlignRightHeadFoot
[short name]  HeadFtR
[active]      yes
[type]        alignment
[definition]  \(o?Xo?\)          $  right  [xo]
```

This regexp matches an X that is enclosed in parentheses with an optional o occurring between the parenthesis and the X. In other words, this regexp matches a trochaic or iambic head foot or a monosyllabic head foot.[18] And, feet with only secondary stress are ignored. AlignRightHeadFoot calculates violations for the given candidate strings as follows:

Candidate	Sum of Alignment Violations
(xo)(Xo)o	1
(ox)(oX)o	1
(xo)(X)o	1
(xo)(Xo)(xo)	2
(ox)(oX)(ox)	2
(xo)(X)(xo)	2
(xo)(Xo)	0
(Xo)(xo)	2
(Xo)(xo)o	3
(Xo)(xo)(xo)	4

Helpful Hint

The question mark ? used in the AlignRightHeadFoot constraint is a regexp metacharacter that indicates optionality. The question mark is placed directly after the part of the regexp that is optional. For example, the regexp ai?ta will match both aita and ata since the presence of the i is optional. The match could be useful as a constraint that penalized all voiceless intervocalic stops and the language allowed both short vowels (a) and diphthongs (ai). In AlignRightHeadFoot, the regexp \(o?Xo?\) will match (X), (oX), (Xo), or (oXo).

[18] Or a trisyllabic amphibrach.

4.4.3 `Scalar`

`Scalar` type constraints assign different numbers of violations for each step in a scale. By default, the violation amount assigned to each step increases linearly in an additive + one fashion although this can be modified via the [`violation pattern`] parameter. Each step in the scale is expressed as a regexp.

The [`constraint`], [`long name`], and [`definition`] parameters are obligatory. And, the [`type`] parameter must be set to `scalar`.

The [`definition`] format of the `scalar` constraint type is the following:

$$\text{step}_{i} \quad \text{step}_{i+1} \quad \text{step}_{i+2} \quad \ldots \quad \text{step}_{i+n}$$

Each step of the scale is separated by a *single tab*. The default order is left to right in increasing violation score. By default $i = 1$, so $step_i$ receives one violation, $step_{i+1}$ receives two violations, $step_{i+2}$ receives three violations, and so forth.

Examples

As an example, a constraint `*C-Nuc` could assign scalar violations to consonantal nuclei for each step along the following sonority scale:

$$\textit{vowel} < \textit{sonorant} < \textit{fricative} < \textit{stop}$$

Since *vowel* is the most sonorous, we simply do not assign any violations to vowels. Here the least sonorous stops would receive the largest violation value and the most sonorous glides would receive the smallest violation value. The schematic representation for the consonantal nuclei could be the following:

$$\textit{sonorant} = \text{R}, \textit{fricative} = \text{F}, \textit{stop} = \text{K}$$

The OT-Help constraint can be defined as follows:

```
[constraint]
[long name]    *C-Nuc
[active]       yes
[type]         scalar
[definition]   R       F   K
```

This `scalar` definition statement assigns one violation for every `R`, two violations for every `F`, and three violations for every `K`. `*C-Nuc` assigns violations to the given candidate strings as follows:

Candidate	Number of Scalar Violations
tA	0
tR	1
tF	2
tK	3
rAtRwRt	2 (= 1 + 1)
Kfr	3
kFr	2
kfR	1
KfR	4 (= 3 + 1)
kFR	3 (= 2 + 1)
KFR	6 (= 3 + 2 + 1)

If the default violation assignment is not desirable, then the user can define the violation score assigned to each step of the scale by hand using the `[violation pattern]` parameter. In this case, the violation arguments of `[violation pattern]` occur in the same order as the arguments of `[definition]` parameter, and the number of arguments for both of these parameters must be identical. Thus, our `*C-Nuc` can be redefined to assign violations exponentially.

```
[constraint]
[long name]          *C-Nuc2
[active]             yes
[type]               scalar
[definition]         R        F    K
[violation pattern]  10       100  1000
```

Our new [violation pattern] in conjunction with the definition assigns ten violations for every R, one hundred violations for every F, and one thousand violations for every K. *C-Nuc2's violation assignments to the same inputs are below:

Input	Number of Scalar Violations
tA	0
tR	10
tF	100
tK	1000
rAtRwRt	20
Kfr	1000
kFr	100
kfR	10
KfR	1010
kFR	110
KFR	1110

Although scalar constraints can assign different numbers of violations to different regexps, this need not be the case.

Technical Note

Scalar Constraints are Flexible

Users may exploit the flexibility of the scalar constraint type to create complex constraints if desired. As an example, alignment constraints only allow one alignment statement in their definition. If a user wanted a constraint to have multiple alignment statements, a scalar constraint could be used for this purpose as long as the user correctly anticipates the maximum counting distances required by the user-chosen inputs. Since scalar constraints were not really designed for this purpose, we leave exploration of this to the fearless user.

4.4.4 Long-distance

Long distance markedness constraints are useful for analyzing consonant harmony and similar phenomena.

The [constraint], [long name], and [definition] parameters are obligatory. And, the [type] parameter must be set to long-distance.

The general definition format of an long-distance constraint is the following

[definition] *L M C Q R*

where *L*, *M*, *C*, *Q*, and *R* are regular expressions for the sets of characters that define:

L = left-hand context
M = what may be skipped over when looking for *L*
C = element counted in determining violations
Q = what may be skipped over when looking for *R*
R = right-hand context

This schema has not yet been tested extensively, so constraints defined in this way should be checked on various inputs before use.

Examples

The following Agr-R constraint assigns a violation mark for every s that is preceded at any distance by an S, provided that no ts intervene. For example, it assigns three marks to both Sasasas and Sasasasatas. Although this constraint only cares about material to the left of the s, the *Q* argument is required and is represented by a period. The *R* argument is missing, but the tab character that precedes it is present.

```
[constraint]
[long name]   Agr-R
[active]      yes
[type]        long-distance
[definition]  S             [^t]  s  .
```

The following Agr-L constraint assigns a violation mark for every s that is followed at any distance by an S, provided that no ts intervene. For example, it assigns three marks

to both sasasaS and satasasasaS. Although this constraint only cares about material to the right of the s, the *M* argument is required and is represented by a period. The *L* argument is missing, but the tab characters that precede and follow it are present.

```
[constraint]
[long name]   Agr-L
[active]      yes
[type]        long-distance
[definition]                .  s  [^t]  S
```

Helpful Hint

The caret ^ used in the Agr-R and Agr-L constraints is a regexp metacharacter that indicates negation within regexp character classes. (More precisely, it indicates a match of the complement set of the set of negated characters.) For example, the [^t] matches any character except t. If more than one character is included in the character class, then all characters following will be negated. For example, [^cTtjDdl] matches any character except c, T, t, j, D, d, and l.

Note that this use of ^ is restricted to within character classes. Outside of character classes, it is the beginning of line metacharacter, which was introduced earlier on page 65.

The next constraint Pseudo-Nati somewhat resembles *nati* in Sanskrit. It assigns a violation mark for every n that is preceded at any distance by S, r, or R, provided that none of the characters — c, T, t, j, D, d, l — intervene, and further provided that n is immediately followed by a character other than k. It assigns two violation marks to Sakamanapankanicena, one for each of the underlined ns.

```
[constraint]
[long name]   Pseudo-Nati
[active]      yes
[type]        long-distance
[definition]  [rRS]          [^cTtjDdl]  n  0  [^k]
```

The `0` (zero) in the *Q* argument of the definition is a trick to ensure that the non-`k` character immediately follows the `n`. The trick is that `0` is not used to represent a sound of this language, so *Q* = `0` is the same as "nothing can intervene between *C* and *R*".

4.4.5 `Faithfulness`

In OT-Help, `faithfulness` type constraints are tied to user-defined operations. This means that faithfulness constraints are "defined" within the operation file and not within the constraint (unlike other user-defined constraints).

When an output candidate is generated via a user-defined operation, then that generated candidate receives a violation for any faithfulness constraints that are specified under that operation's `[violated faith]` parameter. This argument of `[violation faith]` must be the constraint's `[long name]`.

Like other user-defined constraint types, faithfulness constraints are activated from within the constraint file by setting the `active` parameter to `yes`. This means that just associating a faithfulness constraint to an operation within in the operation file is not sufficient for including this faithfulness constraint in typologies: It must also be listed in the constraint file and activated.

The `[constraint]` and `[long name]` parameters are obligatory. And, the `[type]` parameter must be set to `faithfulness`. There is no `[definition]` argument.

Examples

The following `Ident-ATR faithfulness` constraint penalizes input–output mappings that do not correspond in the feature [ATR].

```
[constraint]
[long name]   Ident-ATR
[short name]  Id-ATR
[active]      yes
[type]        faithfulness
```

Since `faithfulness` constraints lack a `[definition]` parameter, we must inspect the associated operation file in order to see how `Ident-ATR` assigns faithfulness violations.

We represent [+ATR] vowels with the character `u` and [−ATR] vowels with `o`. Our associated operation could be the following `ChangeATR`, which changes `o` characters into `u` characters and vice versa.

```
[operation]
[long name]       ChangeATR
[active]          yes
[definition]      u          o
[definition]      o          u
[violated faith]  Ident-ATR
```

The `[violated faith]` parameter of operation `SpreadATR` has the argument `Ident-ATR`, which is the `[long name]` our faithfulness constraint.

Given an input of `tuto`, the `ChangeATR` operation generates the candidates `toto` and `tutu`. Both of these candidates receive one violation of `Ident-ATR`. The other output candidate `tuto`, which is identical to the input, does not violate `Ident-ATR` since it was not generated by any operation.

4.4.6 Positive Constraints

Positive constraints assign positive violation scores to candidates. In other words, they assign rewards. In OT-Help, there is no dedicated `[type]` for positive constraints: any constraint type can become positive via the `[violation pattern]` parameter.

Positive constraints are created by assigning negative numbers to the `[violation pattern]` parameter. The negative sign effectively switches the violation from a penalty to a reward. For instance, if a regexp in the `[definition]` parameter is assigned a `[violation pattern]` of `1`, it assigns a penalty of −1 for every match found in the input. If a regexp in the `[definition]` parameter is assigned a `[violation pattern]` of `-1`, it assigns a reward of +1 for every match found in the input.

Helpful Hint

If you mistakenly try to use the [short name] of a faithfulness constraint as the associated operation's [violated faith] parameter, then the faithfulness constraint and the operation will not be connected to each. In this case, the faithfulness constraint will appear in tableaux in your typology, but it will not assign any violations because it is not associated with any operation. Caution must be exercised so that the [long name] is used for the [violated faith] argument.

Additionally, since [long name] arguments are case-sensitive in OT-Help, you must make sure that the name that occurs in [long name] and [violated faith] is exactly the same. For example, in our Ident-ATR example, if we instead had [violated faith] = Ident-atr in the operation file, then the constraint and the operation would not be associated with each other as Ident-atr and Ident-ATR are different names.

Examples

An example of positive constraint could be PositiveShare. Although feature-spreading constraints have usually been framed in terms of penalties against failures of feature spreading, an alternate formulation could reward any candidate that did spread the feature. Modeling this after the penalizing Share constraint, this positive constraint would be formalized in OT-Help as a regexp that assigns a reward to every pair of segments that is linked to the same feature. Using capital characters to represent segments linked to this feature and lowercase characters for segment pairs that are not linked, the regexp matches every two-character sequence of capital letters. PositiveShare is defined below:

```
[constraint]
[long name]          PositiveShare
[active]             yes
[type]               markedness
[definition]         [A-Z]{2}
[violation pattern]  -1
```

For the regexp, `[A-Z]` is the character class for all capital letters, and the repetition is set to `{2}`. The negative sign of `-1` assigned to this regexp in the `[violation pattern]` parameter indicates that the constraint is positive.

Given the following inputs, `PositiveShare` assigns the following rewards. This rewarding PositiveShare can be compared with the penalizing Share that was discussed on page 61.

Input	Score of Rewards	Substring Match(es)
bayi	0	
Mayi	0	
MAyi	+1	MA
MAYi	+2	MA, AY
MAYI	+3	MA, AY, YI
pukEN	+1	EN

`Scalar` constraints may have multiple regexp steps along a scale as their `[definition]` with arbitrary reward (or penalty) values assigned via the `[violation pattern]` parameter. (See section 4.4.3 for multiple scalar steps in penalizing examples.)

4.4.7 Pre-defined Constraints

A pre-defined constraint is a special type of constraint that is treated differently than other constraints.

It is not defined or included in either the operation or constraint file. Instead, it is simply identified by name in the input file.

Pre-defined constraints are designed to absolve the user from writing complex regexps.[19]

Currently, there is only one pre-defined constraint: `Parse`. This constraint assigns a violation for every unparsed alpha character in the input string.[20] Parsing is represented

[19] Incidentally, OT-Help has no pre-defined operations.
[20] The behavior of this constraint may become more flexible in future versions of OT-Help.

by enclosing one or more alphacharacters in parentheses. So, the pre-defined OT-Help `Parse` assigns a violation to every alphacharacter that is not preceded by an open parens and followed by a closed parens at some point in the string. Given the candidate strings below, pre-defined `Parse` finds the following parsing violations.

Input	Number of Matches	**Substring Match(es)**
(patiku)	0	
(pa)(ti)(ku)	0	
(patik)u	1	a
(pa)tiku	4	t,i,k,u
(pa)(ti)ku	2	k,u
(pa)ti(ku)	2	t,i
(pat)i(k)u	2	i,u

In order to call the pre-defined constraint in the input file, the input file must follow its OTSoft legacy format a little more closely. The input file for the above inputs with the pre-defined `Parse` constraint specified:

```
                        Parse
                        Parse
(patiku)            0   1
(pa)(ti)(ku)        0   1
(patik)u            0   1
(pa)tiku            0   1
(pa)(ti)ku          0   1
(pa)ti(ku)          0   1
(pat)i(k)u          0   1

[end of tableaux]
```

What differs here from a usual input file is the initial two-row preamble. Instead of the `[typology]` and `[begin tableaux]` tags, the first two lines contain the name of the pre-defined constraint, which is preceded by four tabs. The first and second lines should be identical.

Overriding Pre-defined Constraints. Pre-defined constraints are overridden by constraints of the same name that occur within the constraint file. For example, if pre-defined `Parse` is included in the input file and a user-defined `Parse` is included in the constraint, the `Parse` that appears in typologies assigns violations according to the user-defined constraint instead of the pre-defined constraint.

References

Becker, Michael & Pater, Joe. 2007. OT-Help user guide. *University of Massachusetts Occasional Papers in Linguistics* 36: 1–12. http://web.linguist.umass.edu/~OTHelp/OTHelp.pdf

Becker, Michael; Pater, Joe; & Potts, Christopher. 2007. OT-Help 1.2 [Software]. Amherst, MA: University of Massachusetts, Amherst. http://web.linguist.umass.edu/~OTHelp/

Hayes, Bruce; Tesar, Bruce; & Zuraw, Kie. 2003. OTSoft 2.1 [Software]. Los Angeles, CA and New Brunswick, NJ: University of California Los Angeles and Rutgers University. http://www.linguistics.ucla.edu/people/hayes/otsoft/

Legendre, Géraldine; Miyata, Yoshiro; & Smolensky, Paul. 1990a. Harmonic Grammar: A formal multi-level connectionist theory of linguistic well-formedness: Theoretical foundations. In *Proceedings of the 12th Annual Conference of the Cognitive Science Society*, 388–395. Hillsdale: Erlbaum.

Legendre, Géraldine; Miyata, Yoshiro; & Smolensky, Paul. 1990a. Harmonic Grammar: A formal multi-level connectionist theory of linguistic well-formedness: An application. In *Proceedings of the 12th Annual Conference of the Cognitive Science Society*, 884–891. Hillsdale: Erlbaum.

McCarthy, John J. 2002. *A thematic guide to Optimality Theory*. Cambridge: Cambridge University Press.

McCarthy, John J. 2009. Harmony in Harmonic Serialism. Unpublished manuscript, University of Massachusetts, Amherst. ROA-1009: http://roa.rutgers.edu/view.php3?id=1453

McCarthy, John J. to appear. Autosegmental spreading in Optimality Theory. In *Tones and features* (Clements memorial volume), ed. John Goldsmith, Elizabeth Hume, & Leo Wetzels. Berlin: Mouton de Gruyter. http://works.bepress.com/john_j_mccarthy/100/

McCarthy, John J. submitted. An introduction to Harmonic Serialism. *Language and Linguistics Compass*.

McCarthy, John J. & Prince, Alan. 1993. Generalized alignment. In *Yearbook of morphology 1993*, edited by Geert Booij & Jaap van Marle, 79–153. Dordrecht: Kluwer.

Pater, Joe. 2009. Weighted constraints in generative linguistics. *Cognitive Science* 33: 999–1035.

Pater, Joe. to appear. Serial Harmonic Grammar and Berber syllabification. In *Prosody matters: Essays in honor of Elisabeth O. Selkirk*, eds. Toni Borowsky, Shigeto Kawahara, Takahito Shinya & Mariko Sugahara. London: Equinox.

Potts, Christopher; Pater, Joe; Jesney, Karen; Bhatt, Rajesh; & Becker, Michael. 2010. Harmonic Grammar with linear programming: From linear systems to linguistic typology. *Phonology* 27 (1): 77–117.

Prince, Alan. 2002. Arguing optimality. In *University of Massachusetts occasional papers in linguistics 26: Papers in Optimality Theory II*, ed. Angela Carpenter, Andries W. Coetzee, & Paul de Lacy, 269–304. Amherst, MA: Graduate Linguistic Student Association, University of Massachusetts.

Prince, Alan & Smolensky, Paul. 1993/2003. *Optimality theory: Constraint interaction in generative grammar*. Malden, MA: Blackwell.

Prince, Alan & Tesar, Bruce. 2004. Learning phonotactic distributions. In *Constraints in phonological acquisition*, edited by René Kager, Joe Pater, & Wim Zonneveld, 245–291. Cambridge: Cambridge University Press.

Staubs, Robert; Becker, Michael; Potts, Christopher; Pratt, Patrick; McCarthy, John J.; & Pater, Joe. 2009. OT-Help 2.0 [Software]. Amherst, MA: University of Massachusetts, Amherst. http://web.linguist.umass.edu/~OTHelp/

Smolensky, Paul & Legendre, Géraldine, eds. 2006. *The harmonic mind: From neural computation to optimality-theoretic grammar*. 2 vols. Cambridge, MA: MIT Press.

Tesar, Bruce & Smolensky, Paul. 1998. Learnability in Optimality Theory, 29: 229–268.